UNISYS

System V Operating System

User's Guide

Volume 1

January 1988

Printed in U S America
UP–13530

Priced Item

UNISYS

System V Operating System

User's Guide

Volume 1

Unisys is a trademark of Unisys Corporation.

January 1988

Printed in U S America
UP–13530

Priced Item

This document is intended for software releases based on AT&T Release 3 of UNIX System V or a subsequent release of the System unless otherwise indicated.

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places and/or events with the names of any individual living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Dataphone and Teletype are registered trademarks of AT&T. DOCUMENTER'S WORKBENCH is a trademark of AT&T. UNIX is a registered trademark of AT&T in the USA and other countries.
HP is a registered trademark of Hewlett-Packard, Inc.
IBM is a registered trademark of International Business Machines Corporation.

Portions of this material are copyrighted © by
AT&T Technologies
and are reprinted with their permission.

Table of Contents

The following are contained in two volumes.

Preface xiii

Part 1: UNIX System Overview

Chapter 1: What is the UNIX System?

What the UNIX System Does 1-1

How the UNIX System Works 1-3

Chapter 2: Basics for UNIX System Users

Getting Started 2-1

The Terminal 2-2

Obtaining a Login Name 2-13

Establishing Contact with the UNIX System 2-14

Part 2: UNIX System Tutorials

Chapter 3: Using the File System

Introduction 3-1

How the File System is Structured 3-2

Your Place in the File System 3-4

Organizing a Directory 3-16

Accessing and Manipulating Files 3-32

Summary 3-77

Chapter 4: Overview of the Tutorials

Introduction 4-1

Text Editing 4-2

The Shell 4-7

Communicating Electronically 4-12

Programming in the System 4-13

Chapter 5: Line Editor Tutorial (**ed**)

Introducing the Line Editor 5-1

Suggestions for Using this Tutorial 5-3

Getting Started 5-4

Exercise 1 5-15

General Format of **ed** Commands 5-16

Line Addressing 5-17

Exercise 2 5-32

Displaying Text in a File 5-33

Creating Text 5-36

Exercise 3 5-44

Deleting Text 5-46

Substituting Text 5-52

Exercise 4 5-61

Special Characters 5-63

Exercise 5 5-74

Moving Text 5-76

Exercise 6 5-86

Other Useful Commands and Information 5-87

Exercise 7 5-97

Answers to Exercises 5-98

Chapter 6: Screen Editor Tutorial (**vi**)

Introduction 6-1

Getting Started 6-4

Creating a File 6-8

Editing Text: the Command Mode 6-11

Quitting **vi** 6-19

Exercise 1 6-22

Moving the Cursor Around the Screen 6-23

Positioning the Cursor in Undisplayed Text 6-41

Exercise 2 6-53

Creating Text 6-55

Exercise 3 6-60

Deleting Text 6-61

Exercise 4 6-68

Modifying Text 6-69

Cutting And Pasting Text Electronically 6-77

Exercise 5 6-82

Special Commands 6-83

Using Line Editing Commands in **vi** 6-86

Quitting **vi** 6-93

Special Options for **vi** 6-96

Exercise 6 6-99

Answers To Exercises 6-100

Chapter 7: Shell Tutorial

Introduction 7-1

Shell Command Language 7-2

Command Language Exercises 7-37

Shell Programming 7-38

Modifying Your Login Environment 7-91

Shell Programming Exercises 7-98

Answers To Exercises 7-100

Chapter 8: Communication Tutorial

Introduction 8-1

Exchanging Messages 8-2

mail 8-3

mailx 8-16

mailx Overview 8-17

Command Line Options 8-19

How to Send Messages: the Tilde Escapes 8-20

How to Manage Incoming Mail 8-32

The **.mailrc** File 8-42

Sending and Receiving Files 8-47

Networking 8-67

Appendices, Glossary, Index

Appendix A: Summary of the File System

Appendix B: Summary of UNIX System Commands

Appendix C: Quick Reference to **ed** Commands

Appendix D: Quick Reference to **vi** Commands

Appendix E: Summary of Shell Command Language

Appendix F: Setting Up the Terminal

Glossary

Index

List of Figures

Figure 1-1: Model of the UNIX System 1-3
Figure 1-2: Functional View of the Kernel 1-4
Figure 1-3: The Hierarchical Structure of the File System 1-5
Figure 1-4: Example of a File System 1-7
Figure 1-5: Execution of a UNIX System Command 1-12

Figure 2-1: A Video Display Terminal and a Printing Terminal 2-2
Figure 2-2: Keyboard Layout of a Teletype 5410 Terminal 2-5
Figure 2-3: UNIX System Typing Conventions 2-7
Figure 2-4: Data Phone Set, Modem, and Acoustic Coupler 2-16
Figure 2-5: Troubleshooting Problems When Logging In 2-23

Figure 3-1: A Sample File System 3-3
Figure 3-2: Directory of Home Directories 3-5
Figure 3-3: Summary of the **pwd** Command 3-7
Figure 3-4: Full Path Name of the /**user1**/**starship** Directory 3-10
Figure 3-5: Relative Path Name of the **draft** Directory 3-11
Figure 3-6: Relative Path Name from **starship** to **outline** 3-12
Figure 3-7: Example Path Names 3-14
Figure 3-8: Summary of the **mkdir** Command 3-17
Figure 3-9: Descrip. of Output Produced by **ls** −**l** Command 3-25
Figure 3-10: Summary of the **ls** Command 3-26
Figure 3-11: Summary of the **cd** Command 3-29
Figure 3-12: Summary of the **rmdir** Command 3-31
Figure 3-13: Basic Commands for Using Files 3-33
Figure 3-14: Summary of the **cat** Command 3-37
Figure 3-15: Summary of Commands to Use with **pg** 3-39
Figure 3-16: Summary of the **pg** Command 3-43
Figure 3-17: Summary of the **pr** Command 3-47
Figure 3-18: Examples of Teletype Model 40 Line Printers 3-48
Figure 3-19: Summary of the **lp** Command 3-50
Figure 3-20: Summary of the **cp** Command 3-54
Figure 3-21: Summary of the **mv** Command 3-57
Figure 3-22: Summary of the **rm** Command 3-58
Figure 3-23: Summary of the **wc** Command 3-61
Figure 3-24: Summary of the **chmod** Command 3-68
Figure 3-25: Summary of the **diff** Command 3-71
Figure 3-26: Summary of the **grep** Command 3-73
Figure 3-27: Summary of the **sort** Command 3-76

Figure 4-1: Comparison of Line and Screen Editors (**ed** or **vi**) 4-6

Figure 5-1: Summary of **ed** Editor Commands 5-14
Figure 5-2: Summary of Line Addressing 5-31
Figure 5-3: Sample Addresses for Displaying Text 5-34
Figure 5-4: Summary of Commands for Displaying Text 5-35
Figure 5-5: Summary of Commands for Creating Text 5-43
Figure 5-6: Summary of Commands for Deleting Text 5-51
Figure 5-7: Summary of Special Characters 5-73
Figure 5-8: Summary of **ed** Commands for Moving Text 5-85
Figure 5-9: Summary of Other Useful Commands 5-96

Figure 6-1: Displaying a File with a **vi** Window 6-2
Figure 6-2: Summary of Commands for the **vi** Editor 6-21
Figure 6-3: Summary of **vi** Motion Commands (Sheet 1 of 4) 6-37
Figure 6-4: Summary of **vi** Motion Commands (Sheet 2 of 4) 6-38
Figure 6-5: Summary of **vi** Motion Commands (Sheet 3 of 4) 6-39
Figure 6-6: Summary of **vi** Motion Commands (Sheet 4 of 4) 6-40
Figure 6-7: Summary of Additional **vi** Motion Commands 6-52
Figure 6-8: Summary of **vi** Commands for Creating Text 6-59
Figure 6-9: Summary of Delete Commands 6-67
Figure 6-10: Summary of **vi** Commands for Changing Text 6-76
Figure 6-11: Summary of the Yank Command 6-79
Figure 6-12: Summary of **vi** Commands to Cut & Paste Text 6-81
Figure 6-13: Summary of Special Commands 6-85
Figure 6-14: Summary of Line Editor Commands 6-92
Figure 6-15: Summary of the Quit Commands 6-95
Figure 6-16: Summary of Special Options for **vi** 6-98

Figure 7-1: Characters with Special Meanings 7-3
Figure 7-2: Summary of the **echo** Command 7-5
Figure 7-3: Summary of Metacharacters 7-10
Figure 7-4: Summary of the **banner** Command 7-14
Figure 7-5: Summary of the **spell** Command 7-19
Figure 7-6: Summary of the **cut** Command 7-23
Figure 7-7: Summary of the **date** Command 7-25
Figure 7-8: Summary of the **batch** Command 7-28
Figure 7-9: Summary of the **at** Command 7-31
Figure 7-10: Summary of the **ps** Command 7-33
Figure 7-11: Summary of the **kill** Command 7-34

Figure 7-12: Summary of the **nohup** Command 7-35
Figure 7-13: Summary of the **dl** Shell Program 7-42
Figure 7-14: Summary of the **bbday** Command 7-46
Figure 7-15: Summary of the **whoson** Command 7-47
Figure 7-16: Summary of the **get.num** Shell Program 7-49
Figure 7-17: Summary of the **show.param** Shell Program 7-52
Figure 7-18: Summary of the **mknum** Shell Program 7-57
Figure 7-19: Summary of the **num.please** Shell Program 7-57
Figure 7-20: Summary of the **t** Shell Program 7-59
Figure 7-21: Summary of the **log.time** Shell Program 7-61
Figure 7-22: Format of a Here Document 7-63
Figure 7-23: Summary of the **gbday** Command 7-65
Figure 7-24: Summary of the **ch.text** Command 7-67
Figure 7-25: Format of the **for** Loop Construct 7-69
Figure 7-26: Summary of **mv.file** Shell Program 7-72
Figure 7-27: Format of the **while** Loop Construct 7-73
Figure 7-28: Format of the **if...then** Conditional Construct 7-75
Figure 7-29: Format of the **if...then...else** Conditional Const. 7-77
Figure 7-30: Summary of the **search** Shell Program 7-78
Figure 7-31: Summary of the **mv.ex** Shell Program 7-81
Figure 7-32: The **case...esac** Conditional Construct 7-82
Figure 7-33: Summary of the **set.term** Shell Program 7-85
Figure 7-34: Summary of the **tail** Command 7-93

Figure 8-1: Summary of the **mail** Command (sending) 8-8
Figure 8-2: Summary of the **uname** Command 8-11
Figure 8-3: Summary of the **uuname** Command 8-11
Figure 8-4: Summary of the **mail** Command (reading) 8-15
Figure 8-5: Sample **.mailrc** File 8-43
Figure 8-6: Summary of the **uucp** Command 8-58
Figure 8-7: Summary of the **uuto** Command 8-62
Figure 8-8: Summary of the **uustat** Command 8-63
Figure 8-9: Summary of the **uupick** Command 8-66
Figure 8-10: Summary of the **ct** Command 8-70
Figure 8-11: Command Strings for Use with **cu** (Sheet 1 of 2) 8-73
Figure 8-12: Command Strings for Use with **cu** (Sheet 2 of 2) 8-74
Figure 8-13: Summary of the **cu** Command 8-76
Figure 8-14: Summary of the **uux** Command 8-78

Figure A-1: Directory Tree from **root** A-1

Preface

The material in this guide is organized into two major parts: an overview of the UNIX* operating system and a set of tutorials on the main tools available on the UNIX system. A brief description of each part follows. The last section of this Preface, "Notation Conventions," describes the typographical notation with which all the chapters of this *Guide* conform. You may want to refer back to this section from time to time as you read the *Guide*.

System Overview

This part consists of Chapters 1-3, which introduce you to the basic principles of the UNIX operating system. Each chapter builds on information presented in preceding chapters, so it is important to read them in sequence.

- Chapter 1, "What is the UNIX System?", provides an overview of the operating system.
- Chapter 2, "Basics for UNIX System Users," discusses the general rules and guidelines for using the UNIX system. It covers topics related to using your terminal, obtaining a system account, and establishing contact with the UNIX system.
- Chapter 3, "Using the File System," offers a working perspective of the file system. It introduces commands for building your own directory structure, accessing and manipulating the subdirectories and files you organize within it, and examining the contents of other directories in the system for which you have access permission.

*UNIX is a registered trademark of AT&T in the USA and other countries. Portions of the Unisys System V Operating System are derived from the AT&T UNIX V.3 release.

UNIX System Tutorials

The second part of the *Guide* consists of tutorials on the following topics: the **ed** text editor, the **vi** text editor, the shell command language and programming language, and electronic communication tools. For a thorough understanding of the material, we recommend that you work through the examples and exercises as you read each tutorial. The tutorials assume you understand the concepts introduced in Chapters 1-3.

- Chapter 4, "UNIX System Capabilities," introduces the four chapters of tutorials in the second half of the *Guide*. It highlights UNIX system capabilities such as command execution, text editing, electronic communication, programming, and aids to software development.
- Chapter 5, "Line Editor Tutorial (**ed**)," teaches you to how to use the **ed** text editor to create and modify text on a video display terminal or paper printing terminal.
- Chapter 6, "Screen Editor Tutorial (**vi**)," teaches you how to use the visual text editor, **vi**, to create and modify text on a video display terminal.

NOTE: **vi**, the visual editor, is based on software developed by The University of California, Berkeley, California; Computer Science Division, Department of Electrical Engineering and Computer Science, and such software is owned and licensed by the Regents of the University of California.

- Chapter 7, "Shell Tutorial," teaches you to how to use the shell, both as a command interpreter and as a programming language used to create shell programs.
- Chapter 8, "Communication Tutorial," teaches you how to send messages and files to users of both your UNIX system and other UNIX systems.

Reference Information

Six appendices and a glossary of UNIX system terms are also provided for reference.

- Appendix A, "Summary of the File System," illustrates how information is stored in the UNIX operating system.
- Appendix B, "Summary of UNIX System Commands," describes, in alphabetical order, each UNIX system command discussed in the *Guide*.
- Appendix C, "Quick Reference to **ed** Commands," is a quick reference for the line editor, **ed**. (For details, see Chapter 5, "Line Editor Tutorial.") The commands are organized by topic, as they are covered in Chapter 5.
- Appendix D, "Quick Reference to **vi** Commands," is a reference for the full screen editor, **vi**, discussed in Chapter 6, "Screen Editor Tutorial (**vi**)." Commands are organized by topic, as covered in Chapter 6.
- Appendix E, "Summary of Shell Command Language," is a summary of the shell command language, notation, and programming constructs, as discussed in Chapter 7, "Shell Tutorial."
- Appendix F, "Setting Up the Terminal," explains how to configure your terminal for use with the UNIX system, and create multiple windows on the screens of terminals with windowing capability.
- The Glossary defines terms pertaining to the UNIX system used in this book.

Notation Conventions

The following notation conventions are used throughout this *Guide*.

bold	User input, such as commands, options and arguments to commands, variables, and the names of directories and files, appear in **bold**.
italic	Names of variables to which values must be assigned (such as *password*) appear in *italic*.
`constant width`	UNIX system output, such as prompt signs and responses to commands, appear in `constant width`.
< >	Input that does not appear on the screen when typed, such as passwords, tabs, or RETURN, appear between angle brackets.
<^*char*>	Control characters are shown between angle brackets because they do not appear on the screen when typed. The circumflex (^) represents the control key (usually labeled CTRL). To type a control character, hold down the control key while you type the character specified by *char*. For example, the notation <^**d**> means to hold down the control key while pressing the D key; the letter D will not appear on the screen.
[ ]	Command options and arguments that are optional, such as **[-msCj]**, are enclosed in square brackets.
¦	The vertical bar separates optional arguments from which you may choose one. For example, when a command line has the following

format:

command [*arg1* ¦ *arg2*]

You may use either *arg1* or *arg2* when you issue the *command*.

... Ellipses after an argument mean that more than one argument may be used on a single command line.

Arrows on the screen (shown in examples in Chapter 6) represent the cursor.

command(number) A command name followed by a number in parentheses refers to the part of a UNIX system reference manual that documents that command. (There are three reference manuals: the *User's Reference Manual*, *Programmer's Reference Manual*, and *System Administrator's Reference Manual*.) For example, the notation **cat**(1) refers to the page in section 1 (of the *User's Reference Manual*) that documents the **cat** command.

In sample commands the $ sign is used as the shell command prompt. This is not true for all systems. Whichever symbol your system uses, keep in mind that prompts are produced by the system; although a prompt is sometimes shown at the beginning of a command line as it would appear on your screen, you are not meant to type it. (The $ sign is also used to reference the value of positional parameters and named variables; see Chapter 7 for details.)

In all chapters, full and partial screens are used to display examples of how your terminal screen will look when you interact with the UNIX system. These examples show how to use the UNIX system editors, write short programs, and execute commands. The input (characters typed by you) and output (characters printed by the UNIX system) are shown in these screens in accordance with the conventions listed above. All examples apply

regardless of the type of terminal you use.

The commands discussed in each section of a chapter are reviewed at the end of that section. A summary of **vi** commands is found in Appendix D, where they are listed by topic. At the end of some sections, exercises are also provided so you can experiment with the commands. The answers to all the exercises in a chapter are at the end of that chapter.

NOTE: The text in the *User's Guide* was prepared with the UNIX system text editors described in the *Guide* and formatted with the DOCUMENTER'S WORKBENCH Software: **troff**, **tbl**, **pic**, and **mm** macros.

PART 1: UNIX SYSTEM OVERVIEW

Chapter 1: What is the UNIX System?

What the UNIX System Does 1-1

How the UNIX System Works 1-3
- The Kernel 1-4
- The File System 1-5
  - Ordinary Files 1-5
  - Directories 1-6
  - Special Files 1-6
- The Shell 1-8
- Commands 1-9
  - What Commands Do 1-9
  - How to Execute Commands 1-10
  - How Commands Are Executed 1-12

What the UNIX System Does

The UNIX operating system is a set of programs (or software) that controls the computer, acts as the link between you and the computer, and provides tools to help you do your work. It is designed to provide an uncomplicated, efficient, and flexible computing environment. Specifically, the UNIX system offers the following advantages:

- a general purpose system for performing a wide variety of jobs or applications
- an interactive environment that allows you to communicate directly with the computer and receive immediate responses to your requests and messages
- a multi-user environment that allows you to share the computer's resources with other users without sacrificing productivity

  This technique is called timesharing. The UNIX system interacts between users on a rotating basis so quickly that it appears to be interacting with all users simultaneously.
- a multi-tasking environment that enables you to execute more than one program simultaneously.

The organization of the UNIX system is based on four major components:

the kernel	The kernel is a program that constitutes the nucleus of the operating system; it coordinates the functioning of the computer's internals (such as allocating system resources). The kernel works invisibly; you need never be aware of it while doing your work.
the file system	The file system provides a method of handling data that makes it easy to store and access information.

the shell

The shell is a program that serves as the command interpreter. It acts as a liaison between you and the kernel, interpreting and executing your commands. Because it reads input from you and sends you messages, it is described as interactive.

commands

Commands are the names of programs that you request the computer to execute. Packages of programs are called tools. The UNIX system provides tools for jobs such as creating and changing text, writing programs and developing software tools, and exchanging information with others via the computer.

How the UNIX System Works

Figure 1-1 is a model of the UNIX system. Each circle represents one of the main components of the UNIX system: the kernel, the shell, and user programs or commands. The arrows suggest the shell's role as the medium through which you and the kernel communicate. The remainder of this chapter describes each of these components, along with another important feature of the UNIX system, the file system.

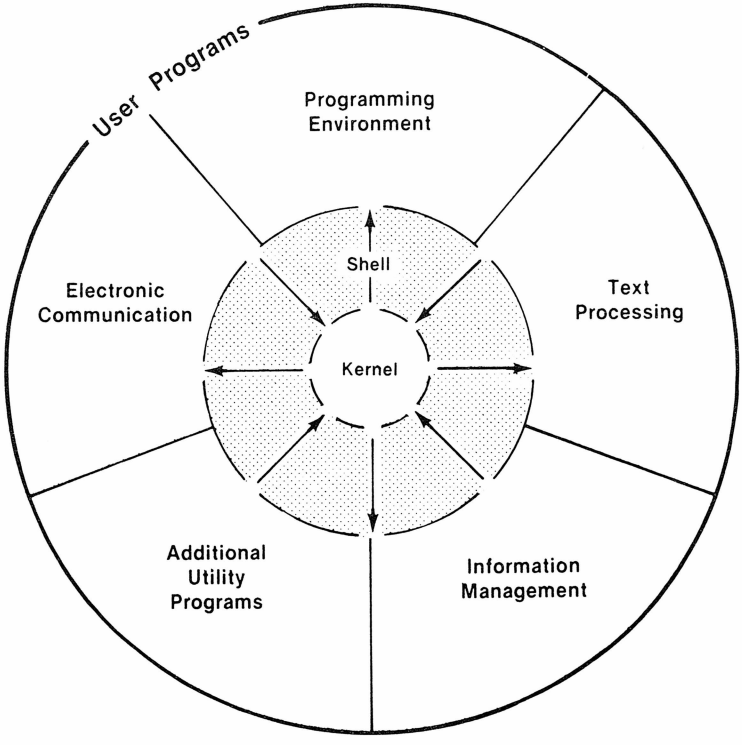


Figure 1-1: Model of the UNIX System

The Kernel

The nucleus of the UNIX system is called the kernel. The kernel controls access to the computer, manages the computer's memory, maintains the file system, and allocates the computer's resources among users. Figure 1-2 is a functional view of the kernel.

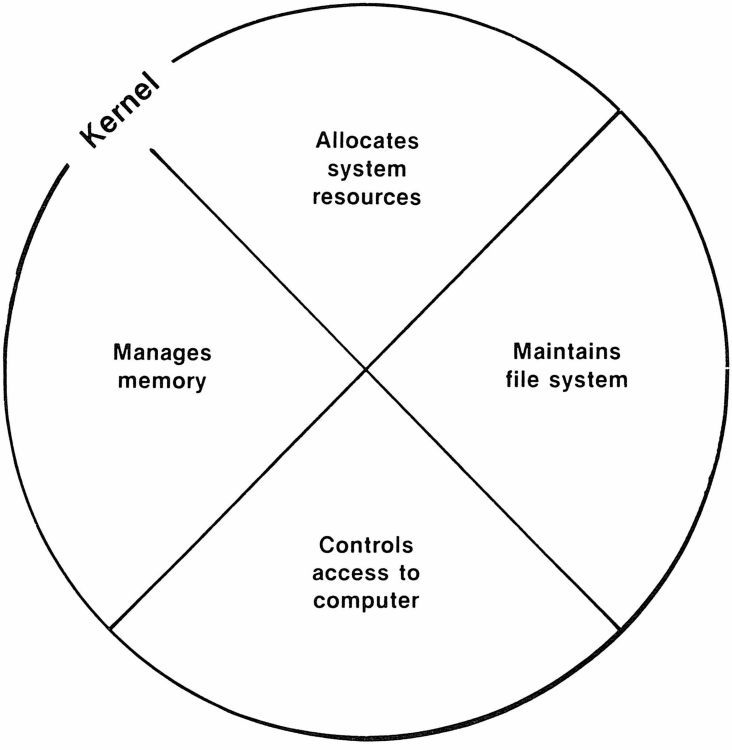


Figure 1-2: Functional View of the Kernel

The File System

The file system is the cornerstone of the UNIX operating system. It provides a logical method of organizing, retrieving, and managing information. The structure of the file system is hierarchical; if you could see it, it might look like an organization chart or an inverted tree (Figure 1-3).

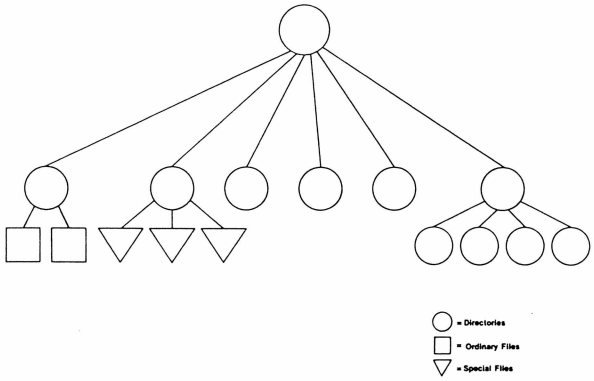


Figure 1-3: The Hierarchical Structure of the File System

The file is the basic unit of the UNIX system and it can be any one of three types: an ordinary file, a directory, or a special file. (See Chapter 3, "Using the File System.")

Ordinary Files

An ordinary file is a collection of characters that is treated as a unit by the system. Ordinary files are used to store any information you want to save. They may contain text for letters or reports, code for the programs you write, or commands to run your programs. Once you have created a file, you can add material to it, delete material from it, or remove it entirely when it is no longer needed.

Directories

A directory is a super-file that contains a group of related files. For example, a directory called **sales** may hold files containing monthly sales figures called **jan**, **feb**, **mar**, and so on. You can create directories, add or remove files from them, or remove directories themselves at any time.

All the directories that you create and own will be located in your home directory. This is a directory assigned to you by the system when you receive a recognized login. You have control over this directory; no one else can read or write files in it without your explicit permission, and you determine its structure.

The UNIX system also maintains several directories for its own use. The structure of these directories is much the same on all UNIX systems. These directories, which include /**unix** (the kernel) and several important system directories, are located directly under the root directory in the file hierarchy. The root directory (designated by /) is the source of the UNIX file structure; all directories and files are arranged hierarchically under it.

Special Files

Special files constitute the most unusual feature of the file system. A special file represents a physical device such as a terminal, disk drive, magnetic tape drive, or communication link. The system reads and writes to special files in the same way it does to ordinary files. However the system's read and write requests do not activate the normal file access mechanism; instead, they activate the device handler associated with the file.

Some operating systems require you to define the type of file you have and to use it in a specified way. In those cases, you must consider how the files are stored since they might be sequential, random-access, or binary files. To the UNIX system, however, all files are alike. This makes the UNIX system file structure easy to use. For example, you need not specify memory requirements for your files since the system automatically does this for you. Or if you or a program you write needs to access a certain device, such as a printer, you specify the device just as you would another one of your files. In the UNIX system, there is only one interface for all input from you and output to you; this

simplifies your interaction with the system.

Figure 1-4 shows an example of a typical file system. Notice that the root directory contains the kernel (/**unix**) and several important system directories.

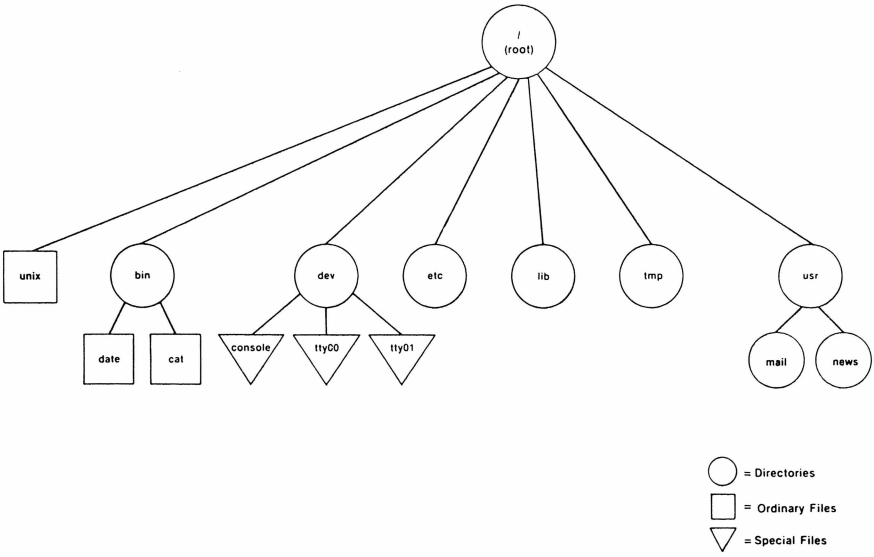


Figure 1-4: Example of a File System

/bin contains many executable programs and utilities

/dev contains special files that represent peripheral devices such as the console, the line printer, user terminals, and disks

/etc	contains programs and data files for system administration
/lib	contains libraries for programs and languages
/tmp	contains temporary files that can be created by any user
/usr	contains other directories including **mail**, which contains files for storing electronic mail, and **news**, which contains files for storing newsworthy items.

In summary, the directories and files you create comprise the portion of the file system that is controlled by you. Other parts of the file system are provided and maintained by the operating system, such as **/bin, /dev, /etc, /lib, /tmp** and **/usr**, and have much the same structure on all UNIX systems.

You will learn more about the file system in other chapters. Chapter 3 shows how to organize a file system directory structure, and access and manipulate files. Chapter 4 gives an overview of UNIX system capabilities. The effective use of these capabilities depends on your familiarity with the file system and your ability to access information stored within it. Chapters 5 and 6 are tutorials designed to teach you how to create and edit files.

The Shell

The shell is a unique command interpreter that allows you to communicate with the operating system. The shell reads the commands you enter and interprets them as requests to execute other programs, access files, or provide output. The shell is also a powerful programming language, not unlike the C programming language, that provides conditional execution and control flow features. The model of a UNIX system in Figure 1-1 shows the two-way flow of communication between you and the computer via the shell.

Chapter 4 describes the shell's capabilities. Chapter 7 is a tutorial that teaches you to write simple shell programs called shell scripts and custom tailor your environment.

Commands

A program is a set of instructions to the computer. Programs that can be executed by the computer without need for translation are called executable programs or commands. As a typical user of the UNIX system, you have many standard programs and tools available to you. If you use the UNIX system to write programs and develop software, you can also draw on system calls, subroutines, and other tools. Of course, any programs you write yourself will be at your disposal, too.

This book introduces you to many of the UNIX system programs and tools that you will use on a regular basis. If you need additional information on these or other standard programs, refer to the *User's Reference Manual*. For information on tools and routines related to programming and software development, consult the *Programmer's Reference Manual*. The *Documentation Roadmap* describes and explains how to order all UNIX system documents from AT&T.

The reference manuals may also be available online. (Online documents are stored in your computer's file system.) You can summon pages from the online manuals by executing the command **man** (short for manual page). For details on how to use the **man** command refer to the **man**(1) page in the *User's Reference Manual*.

What Commands Do

The outer circle of the UNIX system model in Figure 1-1 organizes the system programs and tools into functional categories. These functions include:

text processing	The system provides programs such as line and screen editors for creating and changing text, a spelling checker for locating spelling errors, and optional text formatters for producing high-quality paper copies that are suitable for publication.

information management	The system provides many programs that allow you to create, organize, and remove files and directories.
electronic communication	Several programs, such as **mail**, enable you to transmit information to other users and to other UNIX systems.
software development	Several UNIX system programs establish a friendly programming environment by providing UNIX-to-programming-language interfaces and by supplying numerous utility programs.
additional utilities	The system also offers capabilities for generating graphics and performing calculations.

How to Execute Commands

To make your requests comprehensible to the UNIX system, you must present each command in the correct format, or command line syntax. This syntax defines the order in which you enter the components of a command line. Just as you must put the subject of a sentence before the verb in an English sentence, so must you put the parts of a command line in the order required by the command line syntax. Otherwise, the UNIX system shell will not be able to interpret your request. Here is an example of the syntax of a UNIX system command line.

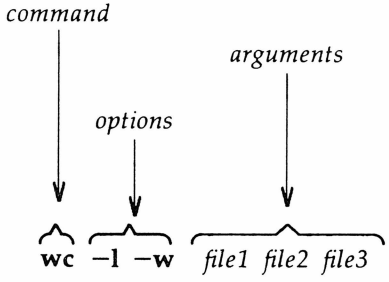
command *option(s)* *argument(s)* <**CR**>

On every UNIX system command line you must type at least two components: a command name and the RETURN key. (The notation <**CR**> is used as an instruction to press the RETURN key throughout this *Guide*.) A command line may also contain either options or arguments, or both. What are commands, options, and arguments?

- a *command* is the name of the program you want to run
- an *option* modifies how the command runs
- an *argument* specifies data on which the command is to operate (usually the name of a directory or file)

In command lines that include options and/or arguments, the component words are separated by at least one blank space. (You can insert a blank by pressing the space bar.) If an argument name contains a blank, enclose that name in double quotation marks. For example, if the argument to your command is **sample 1**, you must type it as follows: "**sample 1**". If you forget the double quotation marks, the shell will interpret **sample** and **1** as two separate arguments.

Some commands allow you to specify multiple options and/or arguments on a command line. Consider the following command line:



In this example, **wc** is the name of the command and two options, **-l** and **-w**, have been specified. (The UNIX system usually allows you to group options such as these to read **-lw** if you prefer.) In addition, three files (*file1*, *file2*, and *file3*) are specified as arguments. Although most options can be grouped together, arguments cannot.

The following examples show the proper sequence and spacing in command line syntax:

Incorrect	Correct
wc*file*	**wc** *file*
wc – l*file*	**wc** **–l** *file*
wc **–l w** *file*	**wc** **–lw** *file*
	or
	wc **–l –w** *file*
wc *file1file2*	**wc** *file1 file2*

Remember, regardless of the number of components, you must end every command line by pressing the RETURN key.

How Commands Are Executed

Figure 1-5 shows the flow of control when the UNIX system executes a command.

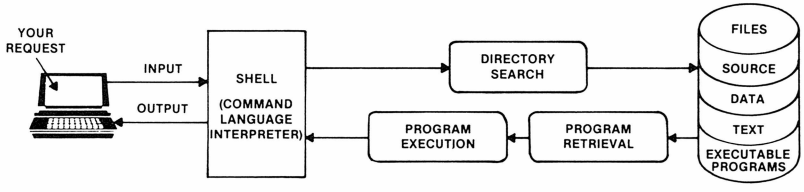


Figure 1-5: Execution of a UNIX System Command

To execute a command, enter a command line when a prompt (such as a $ sign) appears on your screen. The shell considers your command as input, searches through one or more directories to retrieve the program you specified, and conveys your request, along with the program requested, to the kernel. The kernel then follows the instructions in the program and executes the command you requested. After the program has finished running, the shell signals that it is ready for your next command by printing another prompt.

This chapter has described some basic principles of the UNIX operating system. The following chapters will help you apply these principles according to your computing needs.

[This page left blank.]

2. BASICS FOR UNIX SYSTEM USERS

Chapter 2: Basics for UNIX System Users

Getting Started 2-1

The Terminal 2-2
- Required Terminal Settings 2-3
- Keyboard Characteristics 2-4
- Typing Conventions 2-6
  - The Command Prompt 2-8
  - Correcting Typing Errors 2-8
  - Using Special Characters as Literal Characters 2-10
  - Typing Speed 2-11
  - Stopping a Command 2-11
  - Using Control Characters 2-11

Obtaining a Login Name 2-13

Establishing Contact with the UNIX System 2-14
- Login Procedure 2-16
- Password 2-17
- Possible Problems when Logging In 2-21
- Simple Commands 2-23
- The **help** Command 2-24
- Logging Off 2-26

Getting Started

This chapter acquaints you with the general rules and guidelines for working on the UNIX system. Specifically, it lists the required terminal settings, and explains how to use the keyboard, obtain a login, log on and off the system, and enter simple commands.

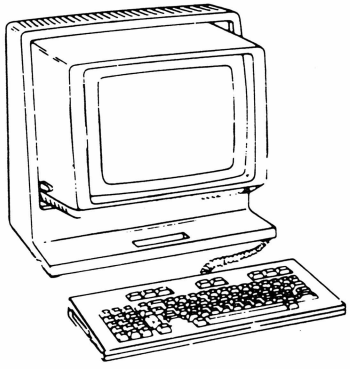
To establish contact with the UNIX system, you need:

- a terminal
- a login name (a name by which the UNIX system identifies you as one of its authorized users)
- a password that verifies your identity
- instructions for dialing in and accessing the UNIX system if your terminal is not directly connected or hard-wired to the computer

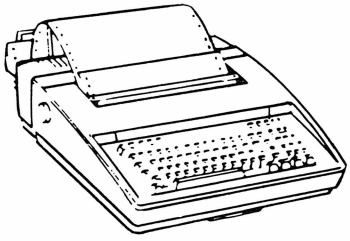
This chapter follows the notation conventions used throughout this *Guide*. For a description of them, see the Preface.

The Terminal

A terminal is an input/output device: you use it to enter requests to the UNIX system, and the system uses it to send its responses to you. There are two basic types of terminals: video display terminals and printing terminals (see Figure 2-1).



Teletype
Model 5410



Teletype
Model 43

Figure 2-1: A Video Display Terminal and a Printing Terminal

The video display terminal shows input and output on a display screen; the printing terminal, on continuously fed paper. In most respects, this difference has no effect on the user's actions or the system's responses. Instructions throughout this book that refer to the terminal screen apply in the same way to the paper in a printing terminal, unless noted otherwise.

Required Terminal Settings

Regardless of the type of terminal you use, you must configure it properly to communicate with the UNIX system. If you have not set terminal options before, you might feel more comfortable seeking help from someone who has.

How you configure a terminal depends on the type of terminal you are using. Some terminals are configured with switches; others are configured directly from the keyboard by using a set of function keys. To determine how to configure your terminal, consult the owner's manual provided by the manufacturer.

The following is a list of configuration checks you should perform on any terminal before trying to log in on the UNIX system.

1. Turn on the power.
2. Set the terminal to ON-LINE or REMOTE operation. This setting ensures the terminal is under the direct control of the computer.
3. Set the terminal to FULL DUPLEX mode. This mode ensures two-way communication (input/output) between you and the UNIX system.
4. If your terminal is not directly connected or hard-wired to the computer, make sure the acoustic coupler or data phone set you are using is set to the FULL DUPLEX mode.
5. Set character generation to LOWER CASE. If your terminal generates only upper case letters, the UNIX system will accommodate it by printing everything in upper case letters.
6. Set the terminal to NO PARITY.
7. Set the baud rate. This is the speed at which the computer communicates with the terminal, measured in characters per second. (For example, a terminal set at a baud rate of 4800 sends and receives 480 characters per second.) Depending on the computer and the terminal, baud rates between 300 and 19200 are available. Some computers may be capable of processing characters at

higher speeds.

Keyboard Characteristics

There is no standard layout for terminal keyboards. However, all terminal keyboards share a standard set of 128 characters called the ASCII character set. (ASCII is an acronym for American Standard Code for Information Interchange.) While the keys are labeled with characters that are meaningful to you (such as the letters of the alphabet), each one is also associated with an ASCII code that is meaningful to the computer.

The keyboard layout on a typical ASCII terminal is basically the same as a typewriter's, with a few additional keys for functions such as interrupting tasks. Figure 2-2 shows an example of a keyboard on an ASCII terminal.

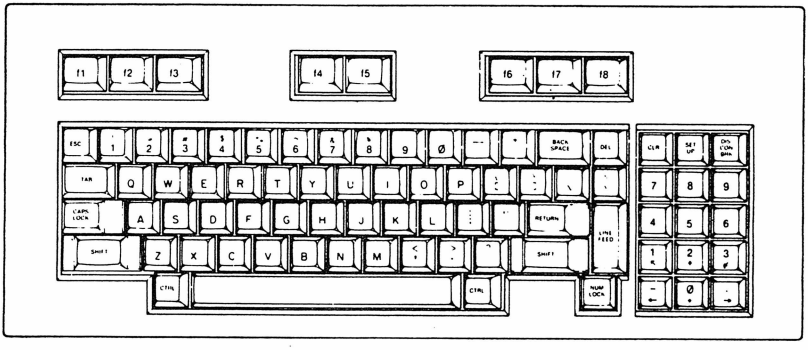


Figure 2-2: Keyboard Layout of a Teletype 5410 Terminal

The keys correspond to the following:

- the letters of the English alphabet (both upper case and lower case)
- the numerals (0 through 9)
- a variety of symbols (including ! @ # $ % ^ & ( ) _ – + = ' { } [ ] \ : ; " ' < > , ? /)
- specially defined words (such as RETURN and BREAK), and abbreviations (such as DEL for delete, CTRL for control, and ESC for escape)

While terminal and typewriter keyboards both have alphanumeric keys, terminal keyboards also have keys designed for use with a computer. These keys are labeled with characters or symbols that remind the user of their functions. However, their placement may vary from terminal to terminal because there is no standard keyboard layout.

Typing Conventions

To interact effectively with the UNIX system, you should be familiar with its typing conventions. The UNIX system requires that you enter commands in lower case letters (unless the command includes an upper case letter). Other conventions enable you to perform tasks, such as erasing letters or deleting lines, simply by pressing one key or entering a specific combination of characters. Characters associated with tasks in this way are known as special characters. Figure 2-3 lists the conventions based on special characters. Detailed explanations of them are provided on the next few pages.

Key(s)	Meaning
$	System's command prompt (your cue to issue a command)
#*	Erase a character
@	Erase or kill an entire line
<BREAK>	Stop execution of a program or command
<DEL>	Delete or kill the current command line
<ESC>	When used with another character, performs a specific function (called an escape sequence)
	When used in an editing session with the **vi** editor, ends the text input mode and returns you to the command mode
<CR>	Press the RETURN key. This ends a line of typing and puts the cursor on a new line.
<^d>†	Stop input to the system or log off
<^h>	Backspace for terminals without a backspace key
<^i>	Horizontal tab for terminals without a tab key
<^s>	Temporarily stops output from printing on the screen
<^q>	Makes the output resume printing on the screen after it has been stopped by the <^s> command

* Nonprinting characters are shown in angle brackets (< >).

† Characters preceded by a circumflex (^) are called control characters. To type a control character, hold down the control key and press the specified letter.

Figure 2-3: UNIX System Typing Conventions

The Command Prompt

The standard UNIX system command prompt is the dollar sign ($). When the prompt appears on your terminal screen, the UNIX system is waiting for instructions from you. The appropriate response to the prompt is to issue a command and press the RETURN key.

The $ sign is the default value for the command prompt. Chapter 7 explains how to change it if you would prefer another character or character string as your command prompt.

Correcting Typing Errors

There are two keys you can use to delete text so that you can correct typing errors. The @ (at) sign key kills the current line and the # (pound) sign key erases the last character typed. These keys are available by default to perform these functions. However, if you want to use other keys, you can reassign the erase and kill functions. (For instructions, see "Reassigning the Delete Functions" later in this section and "Setting Terminal Options" in Chapter 7.)

Deleting the Current Line: the @ Sign

The @ sign key kills the current line. When you press it, an @ sign is added to the end of the line, and the cursor moves to the next line. The line containing the error is not erased from the screen but is ignored.

The @ sign key works only on the current line; be sure to press it before you press the RETURN key if you want to kill a line. In the following example, a misspelled command is typed on a command line; the command is cancelled with the @ sign:

```
whooo@
who<CR>
```

Deleting the Last Characters Typed: the # Sign Key

The # (pound) sign key deletes the character(s) last typed on the current line. When you type a # sign, the cursor backs up over the last character and lets you retype it, thus effectively erasing it. This is an easy way to correct a typing error.

You can delete as many characters as you like as long as you type a corresponding number of # signs. For example, in the following command line, two characters are deleted by typing two # signs.

dattw##e<CR>

The UNIX system interprets this as the **date** command, typed correctly.

The BACKSPACE Key

Many people prefer to use the BACKSPACE key for the erase function instead of the # sign key. When you press the BACKSPACE key, the cursor backs up over your errors, erasing them as it goes. It does not print anything, unlike the # sign key, which prints a # sign on your screen between an error and a correction. When you have finished correcting an error with the BACKSPACE key, the line of text on the screen looks as though it was typed perfectly.

The # sign and BACKSPACE keys are equally effective at deleting characters, but using the BACKSPACE key gives you better visual information about what you are doing.

NOTE: Some terminals may not recognize the # sign key as a delete character.

Reassigning the Delete Functions

As stated earlier, you can change the keys that kill lines and erase characters. If you want to change these keys for a single working session, you can issue a command to the shell to reassign them; the delete functions will revert to the default keys (# and @) as soon as you log off. If you want to use other keys regularly, you must specify the reassignment in a file called **.profile**. Instructions for making both temporary and permanent key

reassignments, along with a description of the **.profile**, are given in Chapter 7.

There are three points to keep in mind if you reassign the delete functions to non-default keys. First, the UNIX system allows only one key at a time to perform a delete function. When you reassign a function to a non-default key, you also take that function away from the default key. For example, if you reassign the erase function from the # sign key to the BACKSPACE key, you will no longer be able to use the # sign key to erase characters. Neither will you have two keys that perform the same function.

Secondly, such reassignments are inherited by any other UNIX system program that allows you to perform the function you have reassigned. For example, the interactive text editor called **ed** (described in Chapter 5) allows you to delete text with the same key you use to correct errors on a shell command line (as described in this section). Therefore, if you reassign the erase function to the BACKSPACE key, you will have to use the BACKSPACE key to erase characters while working in the **ed** editor, as well. The # sign key will no longer work.

Finally, keep in mind that any reassignments you have specified in your **.profile** do not become effective until after you log in. Therefore, if you make an error while typing your login name or password, you must use the # sign key to correct it.

Whichever keys you use, remember that they work only on the current line. Be sure to correct your errors before pressing the RETURN key at the end of a line.

Using Special Characters as Literal Characters

What happens if you want to use a special character in with literal meaning as a unit of text? Since the UNIX system's default behavior is to interpret special characters as commands, you must tell the system to ignore or escape from a character's special meaning whenever you want to use it as a literal character. The backslash (\) enables you to do this. Type a \ before any special character that you want to have treated as it appears. By doing this you essentially tell the system to ignore this character's special meaning and treat it as a literal unit of text.

For example, suppose you want to add the following sentence to a file:

Only one # appears on this sheet of music.

To prevent the UNIX system from interpreting the # sign as a request to delete a character, enter a \ in front of the # sign. If you do not, the system will erase the space after the word one and print your sentence as follows:

```
Only one appears on this sheet of music.
```

To avoid this, type your sentence as follows:

Only one \# appears on this sheet of music.

Typing Speed

After the prompt appears on your terminal screen, you can type as fast as you want, even when the UNIX system is executing a command or responding to one. Since your input and the system's output appear on the screen simultaneously, the printout on your screen will appear garbled. However, while this may be inconvenient for you, it does not interfere with the UNIX system's work because the UNIX system has read-ahead capability. This capability allows the system to handle input and output separately. The system takes and stores input (your next request) while it sends output (its response to your last request) to the screen.

Stopping a Command

If you want to stop the execution of a command, simply press the BREAK or DELETE key. The UNIX system will stop the program and print a prompt on the screen. This is its signal that it has stopped the last command from running and is ready for your next command.

Using Control Characters

Locate the control key on your terminal keyboard. It may be labeled CONTROL or CTRL and is probably to the left of the A key or below the Z key. The control key is used in combination with other characters to perform physical controlling actions on lines of typing. Commands entered in this way are called control

characters. Some control characters perform mundane tasks such as backspacing and tabbing. Others define commands that are specific to the UNIX system. For example, one control character (control-s) temporarily halts output that is being printed on a terminal screen.

To type a control character, hold down the control key and press the appropriate alphabetic key. Most control characters do not appear on the screen when typed and therefore are shown between angle brackets (see "Notation Conventions" in the Preface). The control key is represented by a circumflex (^) before the letter. Thus, for example, <^**s**> designates the control-s character.

The two functions for which control characters are most often used are to control the printing of output on the screen and to log off the system. To prevent information from rolling off the screen on a video display terminal, type <^**s**>; the printing will stop. When you are ready to read more output, type <^**q**> and the printing will resume.

To log off the UNIX system, type <^**d**>. (See "Logging Off" later in this chapter for a detailed description of this procedure.)

In addition, the UNIX system uses control characters to provide capabilities that some terminals fail to make available through function specific keys. If your keyboard does not have a backspace key, you can use the <^**h**> key instead. You can also set tabs without a tab key by typing <^**i**> if your terminal is set properly. (Refer to the section entitled "Possible Problems When Logging In" for information on how to set the tab key.)

Now that you have configured the terminal and inspected the keyboard, one step remains before you can establish communication with the UNIX system: you must obtain a login name.

Obtaining a Login Name

A login name is the name by which the UNIX system verifies that you are an authorized user of the system when you request access to it. It is so called because you must enter it every time you want to log in. (The expression logging in is derived from the fact that the system maintains a log for each user, in which it records the type and amount of system resources being used.)

To obtain a login name, set up a UNIX system account through your local system administrator. There are few rules governing your choice of a login name. Typically, it is three to eight characters long. It can contain any combination of lower case alphanumeric characters, as long as it starts with a letter. It cannot contain any symbols.

However, your login name will probably be determined by local practices. The users of your system may all use their initials, last names, or nicknames as their login names. Here are a few examples of legal login names: **starship**, **mary2**, and **jmrs**.

Establishing Contact with the UNIX System

Typically, you will be using either a terminal that is wired directly to a computer or a terminal that communicates with a computer over a telephone line.

NOTE: This section describes a typical procedure for logging in, but may not apply to your system. There are many ways to log in on a UNIX system over a telephone line. Security precautions on your system may require that you use a special telephone number or other security code. For instructions on logging in on your UNIX system from outside your computer installation site, see your system administrator.

Turn on your terminal. If it is directly connected, the `login:` prompt will immediately appear in the upper lefthand corner of the screen.

If you are going to communicate with the computer over a telephone line, you must now establish a connection. The following procedure is an example of a method you might use to do this. (For the procedure required by your system, see your system administrator.)

1. Dial the telephone number that connects you to the UNIX system. You will hear one of the following:

   A busy signal. This means that either the circuits are busy or the line is in use. Hang up and dial again.

   Continuous ringing and no answer. This usually means that there is trouble with the telephone line or that the system is inoperable because of mechanical failure or electronic problems. Hang up and dial again later.

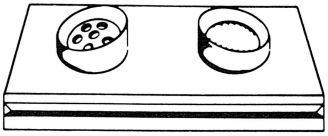
   A high-pitched tone. This means that the system is accessible.

2. When you hear the high-pitched tone, place the handset of the phone in the acoustic coupler or momentarily press the appropriate button on the data phone set (see the owner's manual for the appropriate equipment). Then replace the handset in the cradle (see Figure 2-4).

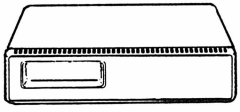
3. After a few seconds, the `login:` prompt will appear in the upper lefthand corner of the screen.

4. A series of meaningless characters may appear on your screen. This means that the telephone number you called serves more than one baud rate; the UNIX system is trying to communicate with your terminal, but is using the wrong speed. Press the BREAK or RETURN key; this signals the system to try another speed. If the UNIX system does not display the `login:` prompt within a few seconds, press the BREAK or RETURN key again.

AT&T Data Phone
Set 212A



AT&T Acoustic
Coupler



AT&T Dataphone II
Modem

Figure 2-4: Data Phone Set, Modem, and Acoustic Coupler

Login Procedure

When the `login:` prompt appears, type your login name and press the RETURN key. For example, if your login name is **starship**, your login line will look like this:

`login:` **starship<CR>**

NOTE: Remember to type in lower case letters. If you use upper case from the time you log in, the UNIX system will expect and respond in upper case exclusively until the next time you log in. It will accept and run many commands typed in upper case, but will not allow you to edit files.

Password

Next, the system prompts you for your password. Type your password and press the RETURN key. For security reasons, the UNIX system does not print (or echo) your password on the screen.

If both your login name and password are acceptable to the UNIX system, the system may print the message of the day and/or current news items and then the default command prompt ($). (The message of the day might include a schedule for system maintenance, and news items might include an announcement of a new system tool.) When you have logged in, your screen will look similar to this:

```
login: starship<CR>
password:
UNIX system news
$
```

If you make a typing mistake when logging in, the UNIX system prints the message `login incorrect` on your screen. Then it gives you a second chance to log in by printing another `login:` prompt.

```
login: ttarship<CR>
password:
login incorrect
login:
```

The login procedure may also fail if the communication link between your terminal and the UNIX system has been dropped. If this happens, you must reestablish contact with the computer (specifically, with the data switch that links your terminal to the computer) before trying to log in again. Since procedures for doing this vary from site to site, ask your system administrator to give you exact instructions for getting a connection on the data switch.

If you have never logged in on the UNIX system, your login procedure may differ from the one just described. This is because some system administrators follow the optional security procedure of assigning temporary passwords to new users when they set up their accounts. If you have a temporary password the system will force you to choose a new password before it allows you to log in.

By forcing you to choose a password for your exclusive use, this extra step helps to ensure a system's security. Protection of system resources and your personal files depends on your keeping your password private.

The actual procedure you follow will be determined by the administrative procedures at your computer installation site. However, it will probably be similar to the following example of a first-time login procedure.

1. You establish contact; the UNIX system displays the `login:` prompt. Type your login name and press the RETURN key.

2. The UNIX system prints the password prompt. Type your temporary password and press the RETURN key.
3. The system tells you your temporary password has expired and you must select a new one.
4. The system asks you to type your old password again. Type your temporary password.
5. The system prompts you to type your new password. Type the password you have chosen.

   Passwords must be constructed to meet the following requirements:

   Each password must have at least six characters. Only the first eight characters are significant.

   Each password must contain at least two alphabetic characters and at least one numeric or special character. Alphabetic characters can be upper case or lower case letters.

   Each password must differ from your login name and any reverse or circular shift of that login name. For comparison purposes, an upper case letter and its corresponding lower case letter are equivalent.

   A new password must differ from the old by at least three characters. For comparison purposes, an upper case letter and its corresponding lower case letter are equivalent.

   Examples of valid passwords are: **mar84ch**, **Jonath0n**, and **BRAV3S**.

NOTE: The UNIX system you are using may have different requirements to consider when choosing a password. Ask your system administrator for details.

6. For verification, the system asks you to reenter your new password. Type your new password again.
7. If you do not reenter the new password exactly as typed the first time, the system tells you the passwords do not match and asks you to try the procedure again. On some systems, however, the communication link may be dropped if you do not reenter the password exactly as typed the first time. If this happens, you must return to step 1 and begin the login procedure again. When the passwords match, the system displays the prompt.

The following screen summarizes this procedure (steps 1 through 6) for first-time UNIX system users.

```
login: starship          <CR>
password: <CR>
Your password has expired.
Choose a new one.
Old password: <CR>
New password: <CR>
Re-enter new password: <CR>
$
```

Possible Problems when Logging In

A terminal usually behaves predictably when you have configured it properly. Sometimes, however, it may act peculiarly. For example, the carriage return may not work properly.

Some problems can be corrected simply by logging off the system and logging in again. If logging in a second time does not remedy the problem, you should first check the following and try logging in once again:

the keyboard	Keys labeled CAPS, LOCAL, BLOCK, and so on should not be enabled (put into the locked position). You can usually disable these keys simply by pressing them.
the data phone set or modem	If your terminal is connected to the computer via telephone lines, verify that the baud rate and duplex settings are correctly specified.
the switches	Some terminals have several switches that must be set to be compatible with the UNIX system. If this is the case with the terminal you are using, make sure they are set properly.

Refer to the section "Required Terminal Settings" in this chapter if you need information to verify the terminal configuration. If you need additional information about the keyboard, terminal, data phone, or modem, check the owner's manuals for the appropriate equipment.

Figure 2-5 presents a list of procedures you can follow to detect, diagnose, and correct some problems you may experience when logging in. If you need further help, contact your system administrator.

Problem†	Possible Cause	Action/Remedy
Meaningless characters	UNIX system at wrong speed	Press RETURN or BREAK key
Input/output appears in UPPER CASE letters	Terminal configuration includes UPPER CASE setting	Log off and set character generation to lower case
Input appears in UPPER CASE, output in lower case	Key labeled CAPS (or CAPS LOCK) is enabled	Press CAPS or CAPS LOCK key to disable setting
Input is printed twice	Terminal is set to HALF DUPLEX mode	Change setting to FULL DUPLEX mode
Tab key does not work properly	Tabs are not set correctly	Type **stty -tabs**‡
Communication link cannot be established although high pitched tone is heard when dialing in	Terminal is set to LOCAL or OFF-LINE mode	Set terminal to ON-LINE mode try logging in again
Communication link (terminal to UNIX system) is repeatedly dropped	Bad telephone line or bad communications port	Call system administrator

* Numerous problems can occur if your terminal is not configured properly. To eliminate these possibilities before attempting to log in, perform the configuration checks listed under "Required Terminal Settings."

† Some problems may be specific to your terminal, data phone set, or modem. Check the owner's manual for the appropriate equipment if suggested actions do not remedy the problem.

‡ Typing **stty -tabs** corrects the tab setting only for your current computing session. To ensure a correct tab setting for all sessions, add the line **stty -tabs** to your **.profile** (see Chapter 7).

Figure 2-5: Troubleshooting Problems When Logging In*

Simple Commands

When the prompt appears on your screen, the UNIX system has recognized you as an authorized user and is waiting for you to request a program by entering a command.

For example, try running the **date** command. After the prompt, type the command and press the RETURN key. The UNIX system accesses a program called **date**, executes it, and prints its results on the screen, as shown below.

```
$ date<CR>
Wed Oct 15 09:49:44 EDT 1986
$
```

As you can see, the **date** command prints the date and time, using the 24-hour clock.

Now type the **who** command and press the RETURN key. Your screen will look something like this:

```
$ who<CR>
starship        tty00       Oct 12      8:53
mary2           tty02       Oct 12      8:56
acct123         tty05       Oct 12      8:54
jmrs            tty06       Oct 12      8:56
$
```

The **who** command lists the login names of everyone currently working on your system. The tty designations refer to the special files that correspond to each user's terminal. The date and time at which each user logged in are also shown.

The help Command

To help you learn how to use these and other commands, the UNIX system provides an on-line teaching aid: the **help** command. This program tells you which command you need to perform a particular task and how to execute specific commands. For novice users of the UNIX system, it also provides definitions of vocabulary and explanations of basic concepts about the system.

NOTE: The **help** command is not available on all UNIX systems; check with your system administrator to find out if it is installed on your system.

When you need assistance, type **help** and press the RETURN key. The program gives you a choice of four ways in which it can help you: by providing general information; by locating the appropriate command for a particular task; by giving you instructions on how to use a particular command; and by defining terms. The following example shows how this menu appears on your screen when you type the command.

```
$ help<CR>
help:   UNIX System On-Line Help


choices         description
s               starter: general information

l               locate: find a command with keywords

u               usage: information about commands

g               glossary: definitions of terms


r               Redirect to a file or a command
q               Quit


Enter choice > _
```

Each choice on this menu (**starter**, **locate**, **usage**, and **glossary**) is an interactive menu program. Request one of these programs by typing the option listed beside it under `choices` (such as **u**).

Because **starter**, **locate**, **usage**, and **glossary** are programs, they can also be called from the shell. Once you are familiar with them, you can skip the step of entering the **help** command first. If you know which program you want to run, you can call it by typing its name as either a command or an argument to the **help**

command. For example, to call the **usage** program, use one of the following command lines:

help usage<CR>

or

usage<CR>

The program you choose responds by printing a summary of its function, a menu of choices, instructions, and examples of how to follow the instructions. In this way, the **help** program leads you through a series of steps that enable you to get the information you need.

Logging Off

When you have completed a session with the UNIX system, type **<^d>** after the prompt. (Remember that control characters such as **<^d>** are typed by holding down the control key and pressing the appropriate alphabetic key. Because they are non-printing characters, they do not appear on your screen.) After several seconds, the UNIX system will display the `login:` prompt again.

```
$ <^d>
login:
```

This shows that you have logged off successfully and the system is ready for someone else to log in.

NOTE: Always log off the UNIX system by typing **<^d>** before you turn off the terminal or hang up the telephone. If you do not, you may not be actually logged off the system.

The **exit** command also allows you to log off but is not used by most users. It may be convenient if you want to include a command to log off within a shell program. (For details, see the "Special Commands" section of the **sh**(1) page in the *User's Reference Manual*.)

3. USING THE FILE SYSTEM

Chapter 3: Using the File System

Introduction 3-1

How the File System is Structured 3-2

Your Place in the File System 3-4
- Your Home Directory 3-4
- Your Current Directory 3-6
- Path Names 3-7
  - Full Path Names 3-8
  - Relative Path Names 3-10
  - Naming Directories and Files 3-15

Organizing a Directory 3-16
- Creating Directories: the **mkdir** Command 3-16
- Listing the Contents of a Directory: the **ls** Command 3-18
- Frequently Used **ls** Options 3-21
- Changing Your Current Directory: the **cd** Command 3-27
- Removing Directories: the **rmdir** Command 3-29

Accessing and Manipulating Files 3-32
- Basic Commands 3-32
  - Displaying a File's Contents: **cat**, **pg**, and **pr** 3-34
  - Requesting a Paper Copy of a File: the **lp** Command 3-48
  - Making a Duplicate Copy of a File: the **cp** Command 3-51
  - Moving and Renaming a File: the **mv** Command 3-54
  - Removing a File: the **rm** Command 3-57
  - Counting Lines, Words, and Characters in a File: the **wc** Command 3-59
  - Protecting Your Files: the **chmod** Command 3-62

Advanced Commands 3-69
Identifying Differences Between Files: the **diff** Command 3-69
Searching a File for a Pattern: the **grep** Command 3-71
Sorting and Merging Files: the **sort** Command 3-73

Summary 3-77

Introduction

To use the UNIX file system effectively you must be familiar with its structure, know something about your relationship to this structure, and understand how the relationship changes as you move around within it. This chapter prepares you to use this file system.

The first two sections ("How the File System is Structured" and "Your Place in the File System") offer a working perspective of the file system. The rest of the chapter introduces UNIX system commands that allow you to build your own directory structure, access and manipulate the subdirectories and files you organize within it, and examine the contents of other directories in the system for which you have access permission.

Each command is discussed in a separate subsection. Tables at the end of these subsections summarize the features of each command so that you can later review a command's syntax and capabilities quickly. Many of the commands presented in this section have additional, sophisticated uses. These, however, are left for more experienced users and are described in other UNIX system documentation. All the commands presented here are basic to using the file system efficiently and easily. Try using each command as you read about it.

How the File System is Structured

The file system is comprised of a set of ordinary files, special files, and directories. These components provide a way to organize, retrieve, and manage information electronically. Chapter 1 introduced the properties of directories and files; this section will review them briefly before discussing how to use them.

- An ordinary file is a collection of characters stored on a disk. It may contain text for a report or code for a program.
- A special file represents a physical device, such as a terminal or disk.
- A directory is a collection of files and other directories (sometimes called subdirectories). Use directories to group files together on the basis of any criteria you choose. For example, you might create a directory for each product that your company sells or for each of your student's records.

The set of all the directories and files is organized into a tree shaped structure. Figure 3-1 shows a sample file structure with a directory called root (/) as its source. By moving down the branches extending from root, you can reach several other major system directories. By branching down from these, you can, in turn, reach all the directories and files in the file system.

In this hierarchy, files and directories that are subordinate to a directory have what is called a parent/child relationship. This type of relationship is possible for many layers of files and directories. In fact, there is no limit to the number of files and directories you may create in any directory that you own. Neither is there a limit to the number of layers of directories that you may create. Thus you have the capability to organize your files in a variety of ways, as shown in Figure 3-1.

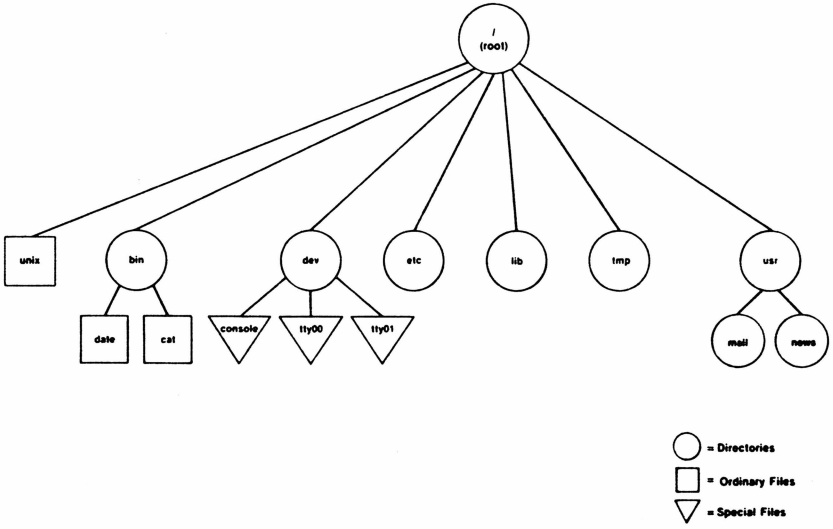


Figure 3-1: A Sample File System

Your Place in the File System

Whenever you interact with the UNIX system, you do so from a location in its file system structure. The UNIX system automatically places you at a specific point in its file system every time you log in. From that point, you can move through the hierarchy to work in any of your directories and files and to access those belonging to others that you have permission to use.

The following sections describe your position in the file system structure and how this position changes as you move through the file system.

Your Home Directory

When you successfully complete the login procedure, the UNIX system places you at a specific point in its file system structure called your login or home directory. The login name assigned to you when your UNIX system account was set up is usually the name of this home directory. Every user with an authorized login name has a unique home directory in the file system.

The UNIX system is able to keep track of all these home directories by maintaining one or more system directories that organize them. For example, the home directories of the login names **starship**, **mary2**, and **jmrs** are contained in a system directory called **user1**. Figure 3-2 shows the position of a system directory such as **user1** in relation to the other important UNIX system directories discussed in Chapter 1.

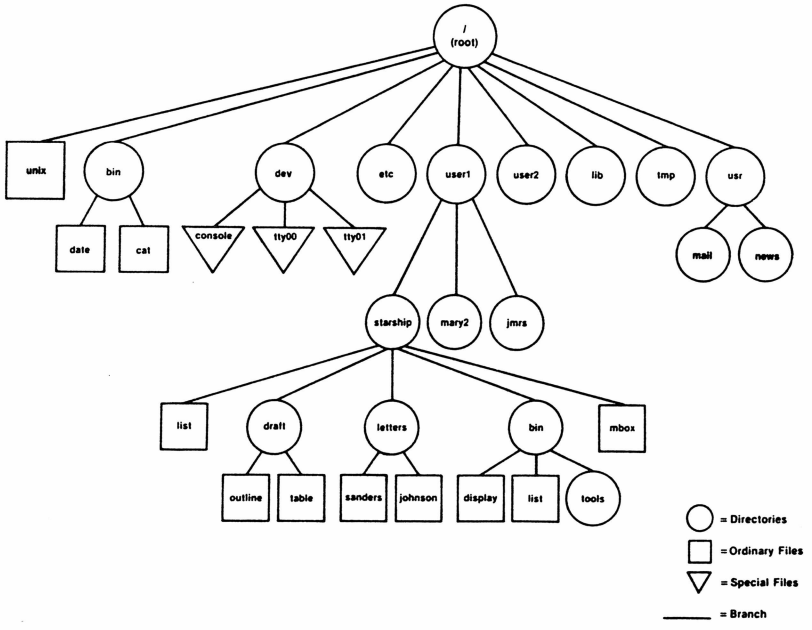


Figure 3-2: Directory of Home Directories

Within your home directory, you can create files and additional directories (sometimes called subdirectories) in which to group them. You can move and delete your files and directories, and you can control access to them. You have full responsibility for everything you create in your home directory because you own it. Your home directory is a vantage point from which to view all the files and directories it holds, and the rest of the file system, all the way up to root.

Your Current Directory

As long as you continue to work in your home directory, it is considered your current working directory. If you move to another directory, that directory becomes your new current directory.

The UNIX system command **pwd** (short for print working directory) prints the name of the directory in which you are now working. For example, if your login name is **starship** and you execute the **pwd** command in response to the first prompt after logging in, the UNIX system will respond as follows:

```
$ pwd<CR>
/user1/starship
$
```

The system response gives you both the name of the directory in which you are working (**starship**) and the location of that directory in the file system. The path name `/user1/starship` tells you that the root directory (shown by the leading / in the line) contains the directory **user1** which, in turn, contains the directory **starship**. (All other slashes in the path name other than root are used to

separate the names of directories and files, and to show the position of each directory relative to root.) A directory name that shows the directory's location in this way is called a full or complete directory name or path name. In the next few pages we will analyze and trace this path name so you can start to move around in the file system.

Remember, you can determine your position in the file system at any time simply by issuing a **pwd** command. This is especially helpful if you want to read or copy a file and the UNIX system tells you the file you are trying to access does not exist. You may be surprised to find you are in a different directory than you thought.

Figure 3-3 provides a summary of the syntax and capabilities of the **pwd** command.

Command Recap **pwd** - print full name of working directory		
command	*options*	*arguments*
pwd	none	none
Description:	**pwd** prints the full path name of the directory in which you are currently working.	

Figure 3-3: Summary of the **pwd** Command

Path Names

Every file and directory in the UNIX system is identified by a unique path name. The path name shows the location of the file or directory, and provides directions for reaching it. Knowing how to follow the directions given by a path name is your key to moving around the file system successfully. The first step in learning about these directions is to learn about the two types of path

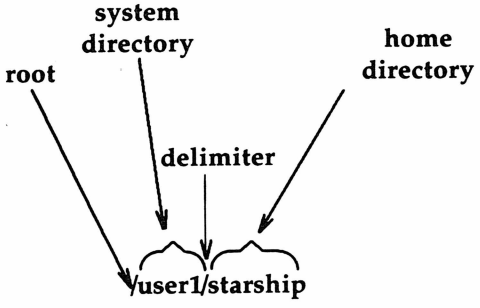
names: full and relative.

Full Path Names

A full path name (sometimes called an absolute path name) gives directions that start in the root directory and lead you down through a unique sequence of directories to a particular directory or file. You can use a full path name to reach any file or directory in the UNIX system in which you are working.

Because a full path name always starts at the root of the file system, its leading character is always a / (slash). The final name in a full path name can be either a file name or a directory name. All other names in the path must be directories.

To understand how a full path name is constructed and how it directs you, consider the following example. Suppose you are working in the **starship** directory, located in **/user1**. You issue the **pwd** command and the system responds by printing the full path name of your working directory: `/user1/starship`. Analyze the elements of this path name using the following diagram and key.



/ (leading)	the slash that appears as the first character in the path name is the root of the file system
user1	system directory one level below root in the hierarchy to which root points or branches
/ (subsequent)	the next slash separates or delimits the directory names **user1** and **starship**
starship	current working directory

Now follow the bold lines in Figure 3-4 to trace the full path to **/user1/starship**.

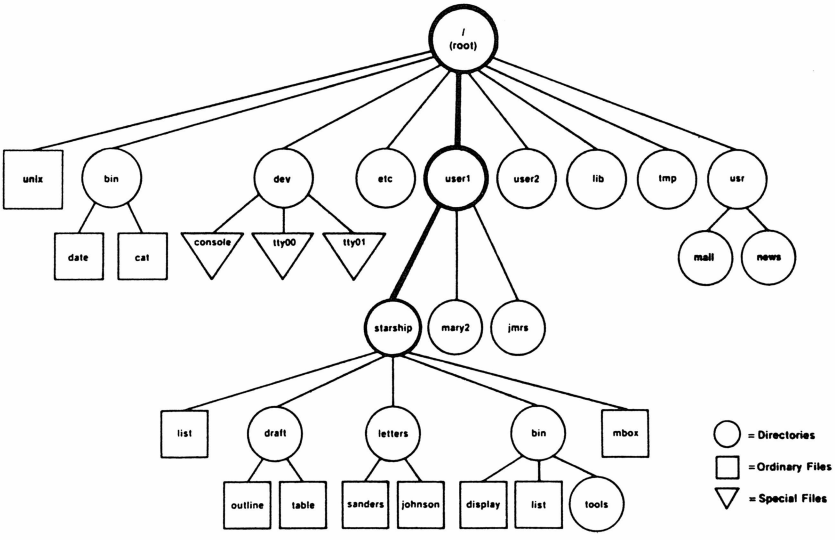


Figure 3-4: Full Path Name of the **/user1/starship** Directory

Relative Path Names

A relative path name gives directions that start in your current working directory, and lead you up or down through a series of directories to a particular file or directory. By moving down from your current directory, you can access files and directories you own. By moving up from your current directory, you pass through layers of parent directories to the grandparent of all system directories, root. From there you can move anywhere in the file system.

A relative path name begins with one of the following: a directory or file name; a **.** (pronounced dot), which is a shorthand notation for your current directory; or a **..** (pronounced dot dot), which is a shorthand notation for the directory immediately above your current directory in the file system hierarchy. The directory represented by **..** (dot dot) is called the parent directory of **.** (your current directory).

For example, say you are in the directory **starship** in the sample system and **starship** contains directories named **draft**, **letters**, and **bin** and a file named **mbox**. The relative path name to any of these is simply its name, such as **draft** or **mbox**. Figure 3-5 traces the relative path from **starship** to **draft**.

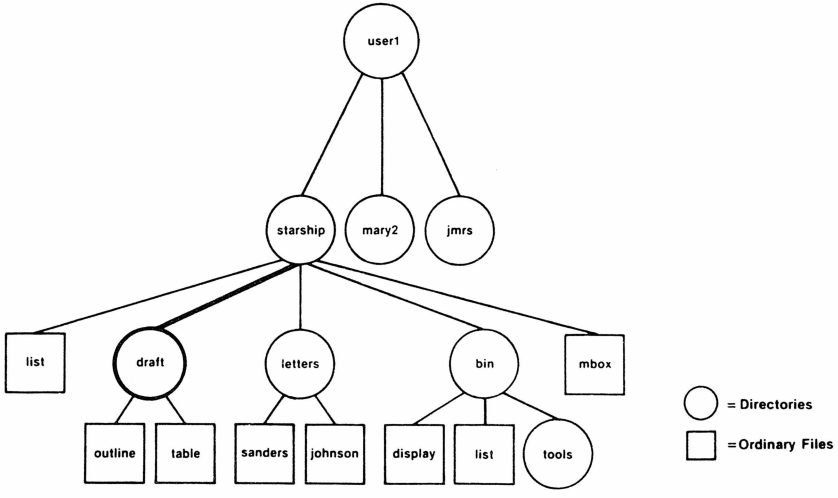


Figure 3-5: Relative Path Name of the **draft** Directory

The **draft** directory belonging to **starship** contains the files **outline** and **table**. The relative path name from **starship** to the file **outline** is **draft/outline**.

Figure 3-6 traces this relative path. Notice that the slash in this path name separates the directory named **draft** from the file named **outline**. Here, the slash is a delimiter showing that **outline** is subordinate to **draft**; that is, **outline** is a child of its parent, **draft**.

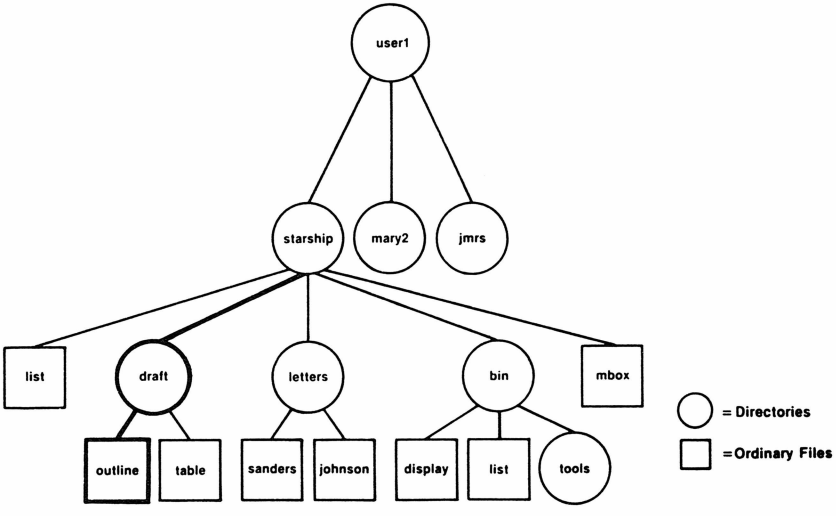


Figure 3-6: Relative Path Name from starship to outline

So far, the discussion of relative path names has covered how to specify names of files and directories that belong to, or are children of, your current directory. You now know how to move down the system hierarchy level by level until you reach your destination. You can also, however, ascend the levels in the system structure or ascend and subsequently descend into other files and directories.

To ascend to the parent of your current directory, you can use the **..** notation. This means that if you are in the directory named **draft** in the sample file system, **..** is the path name to **starship**, and **../..** is the path name to **starship**'s parent directory, **user1**.

From **draft**, you can also trace a path to the file **sanders** by using the path name **../letters/sanders**. The **..** brings you up to **starship**. Then the names **letters** and **sanders** take you down through the **letters** directory to the **sanders** file.

Keep in mind that you can always use a full path name in place of a relative one.

Figure 3-7 shows some examples of full and relative path names.

Path Name	Meaning
/	full path name of the root directory
/bin	full path name of the **bin** directory (contains most executable programs and utilities)
/user1/starship/bin/tools	full path name of the **tools** directory belonging to the **bin** directory that belongs to the **starship** directory belonging to **user1** that belongs to root
bin/tools	relative path name to the file or directory **tools** in the directory **bin** If the current directory is /, then the UNIX system searches for **/bin/tools**. However, if the current directory is **starship**, then the system searches the full path **/user1/starship/bin/tools**.
tools	relative path name of a file or directory **tools** in the current directory.

Figure 3-7: Example Path Names

You may need some practice before you can use path names such as these to move around the file system with confidence. However, this is to be expected when learning a new concept.

Naming Directories and Files

You can give your directories and files any names you want, as long as you observe the following rules:

- The name of a directory (or file) can be from one to fourteen characters long.
- All characters other than / are legal.
- Some characters are best avoided, such as a space, tab, backspace, and the following:

  ? @ # $ ^ & * ( ) ` [ ] \ ¦ ; ' " < >

  If you use a blank or tab in a directory or file name, you must enclose the name in quotation marks on the command line.
- Avoid using a **,** **-** or **.** as the first character in a file name.
- Upper case and lower case characters are distinct to the UNIX system. For example, the system considers a directory (or file) named **draft** to be different from one named **DRAFT**.

The following are examples of legal directory or file names:

memo	**MEMO**	**section2**	**ref:list**
file.d	**chap3+4**	**item1-10**	**outline**

The rest of this chapter introduces UNIX system commands that enable you to examine the file system.

Organizing a Directory

This section introduces four UNIX system commands that enable you to organize and use a directory structure: **mkdir**, **ls**, **cd**, and **rmdir**.

mkdir	enables you to make new directories and subdirectories within your current directory
ls	lists the names of all the subdirectories and files in a directory
cd	enables you to change your location in the file system from one directory to another
rmdir	enables you to remove an empty directory

These commands can be used with either full or relative path names. Two of the commands, **ls** and **cd**, can also be used without a path name. Each command is described more fully in the four sections that follow.

Creating Directories: the mkdir Command

It is recommended that you create subdirectories in your home directory according to a logical and meaningful scheme that will facilitate the retrieval of information from your files. If you put all files pertaining to one subject together in a directory, you will know where to find them later.

To create a directory, use the command **mkdir** (short for make directory). Simply enter the command name, followed by the name you are giving your new directory or file. For example, in the sample file system, the owner of the **draft** subdirectory created **draft** by issuing the following command from the home directory (**/user1/starship**):

```
$ mkdir draft <CR>
$
```

The second prompt shows that the command has succeeded; the subdirectory **draft** has been created.

Still in the home directory, this user created other subdirectories, such as **letters** and **bin**, in the same way.

```
$ mkdir letters <CR>
$ mkdir bin <CR>
$
```

The user could have created all three subdirectories (**draft**, **letters**, and **bin**) simultaneously by listing them all on a single command line.

```
$ mkdir draft letters bin <CR>
$
```

You can also move to a subdirectory you created and build additional subdirectories within it. When you build directories or create files, you can name them anything you want as long as you follow the guidelines listed earlier under "Naming Directories and Files."

Figure 3-8 summarizes the syntax and capabilities of the **mkdir** command.

Command Recap **mkdir** - make a new directory		
command	*options*	*arguments*
mkdir	none	*directoryname(s)*
Description:	**mkdir** creates a new directory (subdirectory).	
Remarks:	The system returns a prompt ($ by default) if the directory is successfully created.	

Figure 3-8: Summary of the **mkdir** Command

Listing the Contents of a Directory: the ls Command

All directories in the file system have information about the files and directories they contain, such as name, size, and the date last modified. You can obtain this information about the contents of your current directory and other system directories by executing the command **ls** (short for list).

The **ls** command lists the names of all files and subdirectories in a specified directory. If you do not specify a directory, **ls** lists the names of files and directories in your current directory. To understand how the **ls** command works, consider the sample file system (Figure 3-2) once again.

Say you are logged in to the UNIX system and you run the **pwd** command. The system responds with the path name **/user1/starship**. To display the names of files and directories in this current directory, you then type **ls** and press the RETURN key. After this sequence, your terminal will read:

```
$ pwd<CR>
$/user1/starship
$ ls<CR>
bin
draft
letters
list
mbox
$
```

As you can see, the system responds by listing, in alphabetical order, the names of files and directories in the current directory **starship**. (If the first character of any of the file or directory names had been a number or an upper case letter, it would have been printed first.)

To print the names of files and subdirectories in a directory other than your current directory without moving from your current directory, you must specify the name of that directory as follows:

ls *pathname* **<CR>**

The directory name can be either the full or relative path name of the desired directory. For example, you can list the contents of **draft** while you are working in **starship** by entering **ls draft** and pressing the RETURN key. Your screen will look like this:

```
$ ls draft<CR>
outline
table
$
```

Here, **draft** is a relative path name from a parent (**starship**) to a child (**draft**) directory.

You can also use a relative path name to print the contents of a parent directory when you are located in a child directory. The **..** (dot dot) notation provides an easy way to do this. For example, the following command line specifies the relative path name from **starship** to **user1**:

```
$ ls ..<CR>
jmrs
mary2
starship
$
```

You can get the same results by using the full path name from root to **user1**. If you type **ls /user1** and press the RETURN key, the system will respond by printing the same list.

Similarly, you can list the contents of any system directory that you have permission to access by executing the **ls** command with a full or relative path name.

The **ls** command is useful if you have a long list of files and you are trying to determine whether one of them exists in your current directory. For example, if you are in the directory **draft** and you want to determine if the files named **outline** and **notes** are there, use the **ls** command as follows:

```
$ ls outline notes<CR>
outline
notes not found
$
```

The system acknowledges the existence of **outline** by printing its name, and says that the file **notes** is not found.

The **ls** command does not print the contents of a file. If you want to see what a file contains, use the **cat**, **pg**, or **pr** command. These commands are described in "Accessing and Manipulating Files," later in this chapter.

Frequently Used ls Options

The **ls** command also accepts options that cause specific attributes of a file or subdirectory to be listed. There are more than a dozen available options for the **ls** commands. Of these, the **–a** and **–l** will probably be most valuable in your basic use of the UNIX system. Refer to the **ls**(1) page in the *User's Reference Manual* for details about other options.

Listing All Names in a File

Some important file names in your home directory, such as **.profile** (pronounced dot-profile), begin with a period. (As you can see from this example, when a period is used as the first character of a file name it is pronounced dot.) When a file name begins with a dot, it is not included in the list of files reported by the **ls** command. If you want the **ls** to include these files, use the **-a** option on the command line.

For example, to list all the files in your current directory (**starship**), including those that begin with a **.** (dot), type **ls –a** and press the RETURN key.

```
$ ls -a<CR>
.
..
.profile
bin
draft
letters
list
mbox
$
```

Listing Contents in Short Format

The **-C** and **-F** options for the **ls** command are frequently used. Together, these options list a directory's subdirectories and files in columns, and identify executable files (with an *) and directories (with a /). Thus, you can list all files in your working directory **starship** by executing the command line shown here:

```
$ ls -CF<CR>
bin/            letters/        mbox
draft/          list*
$
```

Listing Contents in Long Format

Probably the most informative **ls** option is **-l**, which displays the contents of a directory in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file. For example, say you run the **ls -l** command while in the **starship** directory.

```
$ ls -l<CR>
total 30
drwxr-xr-x  3 starship  project  96  Oct 27  08:16  bin
drwxr-xr-x  2 starship  project  64  Nov  1  14:19  draft
drwxr-xr-x  2 starship  project  80  Nov  8  08:41  letters
-rwx------  2 starship  project  31  Nov  2  10:15  list
-rw-------  1 starship  project  40  Oct 27  10:00  mbox
$
```

The first line of output (`total 30`) shows the amount of disk space used, measured in blocks. Each of the rest of the lines comprises a report on a directory or file in **starship**. The first character in each line (`d`, `-`, `b`, or `c`) tells you the type of file.

`d`	directory
`-`	ordinary disk file
`b`	block special file
`c`	character special file

Using this key to interpret the previous screen, you can see that the **starship** directory contains three directories and two ordinary disk files.

The next several characters, which are either letters or hyphens, identify who has permission to read and use the file or directory. (Permissions are discussed in the description of the **chmod** command under "Accessing and Manipulating Files" later in this chapter.)

The following number is the link count. For a file, this equals the number of users linked to that file. For a directory, this number shows the number of directories immediately under it plus two (for the directory itself and its parent directory).

Next, the login name of the file's owner appears (here it is **starship**), followed by the group name of the file or directory (**project**).

The following number shows the length of the file or directory entry measured in units of information (or memory) called bytes. The month, day, and time that the file was last modified is given next. Finally, the last column shows the name of the directory or file.

Figure 3-9 identifies each column in the rows of output from the **ls -l** command.

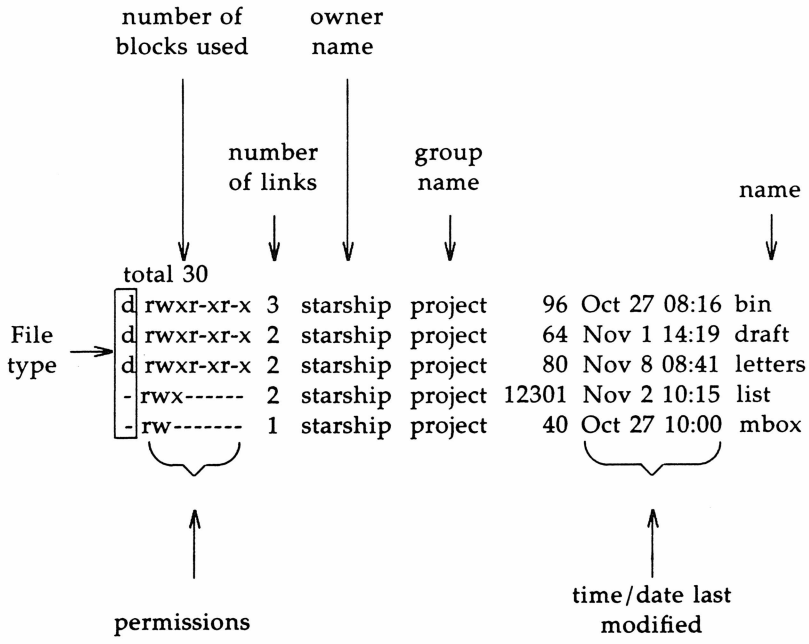


Figure 3-9: Description of Output Produced by the **ls -l** Command

Figure 3-10 summarizes the syntax and capabilities of the **ls** command and two available options.

Command Recap **ls** - list contents of a directory		
command	*options*	*arguments*
ls	**-a, -l,** and others*	*directoryname(s)*
Description:	**ls** lists the names of the files and subdirectories in the specified directories. If no directory name is given as an argument, the contents of your working directory are listed.	
Options:	**-a** Lists all entries, including those beginning with . (dot). **-l** Lists contents of a directory in long format furnishing mode, permissions, size in bytes, and time of last modification.	
Remarks:	If you want to read the contents of a file, use the **cat** command.	

* See the **ls**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-10: Summary of the **ls** Command

Changing Your Current Directory: the cd Command

When you first log in on the UNIX system, you are placed in your home directory. As long as you do work in it, it is also your current working directory. However, by using the command **cd** (short for change directory), you can work in other directories as well. To use this command, enter **cd**, followed by a path name to the directory to which you want to move.

cd *pathname_of_newdirectory*<CR>

Any valid path name (full or relative) can be used as an argument to the **cd** command. If you do not specify a path name, the command will move you to your home directory. Once you have moved to a new directory, it becomes your current directory.

For example, to move from the **starship** directory to its child directory **draft** (in the sample file system), type **cd draft** and press the RETURN key. (Here **draft** is the relative path name to the desired directory.) When you get a prompt, verify your new location by typing **pwd** and pressing the RETURN key. Your terminal screen will look like this:

```
$ cd draft<CR>
$ pwd<CR>
/user1/starship/draft
$
```

Now that you are in the **draft** directory you can create subdirectories in it by using the **mkdir** command, and new files, by using the **ed** and **vi** editors. (See Chapters 5 and 6 for tutorials on the **ed** and **vi** commands, respectively.)

It is not necessary to be in the **draft** directory to access files within it. You can access a file in any directory by specifying a full path name for it. For example, to **cat** the **sanders** file in the **letters** directory (/**user1**/**starship**/**letters**) while you are in the **draft** directory (/**user1**/**starship**/**draft**), specify the full path name of **sanders** on the command line.

cat /user1/starship/letters/sanders <CR>

You may also use full path names with the **cd** command. For example, to move to the **letters** directory from the **draft** directory, specify /**user1**/**starship**/**letters** on the command line, as follows:

cd /user1/starship/letters <CR>

Also, because **letters** and **draft** are both children of **starship**, you can use the relative path name **../letters** with the **cd** command. The **..** notation moves you to the directory **starship**, and the rest of the path name moves you to **letters**.

Figure 3-11 summarizes the syntax and capabilities of the **cd** command.

Command Recap **cd** - change your working directory		
command	*options*	*arguments*
cd	none	*directoryname*
Description:	**cd** changes your position in the file system from the current directory to the directory specified. If no directory name is given as an argument, the **cd** command places you in your home directory.	
Remarks:	When the shell places you in the directory specified, the prompt ($ by default) is returned to you. To access a directory that is not in your working directory, you must use the full or relative path name in place of a simple directory name.	

Figure 3-11: Summary of the **cd** Command

Removing Directories: the rmdir Command

If you no longer need a directory, you can remove it with the command **rmdir** (short for remove a directory). The standard syntax for this command is:

rmdir *directoryname(s)* <CR>

You can specify more than one directory name on the command line.

The **rmdir** command will not remove a directory if you are not the owner of it or if the directory is not empty. If you want to remove a file in another user's directory, the owner must give you write permission for the parent directory of the file you want to remove.

If you try to remove a directory that still contains subdirectories and files (that is, is not empty), the **rmdir** command prints the message *directoryname* `not empty`. You must remove all subdirectories and files; only then will the command succeed.

For example, say you have a directory called **memos** that contains one subdirectory, **tech**, and two files, **june.30** and **july.31**. (Create this directory in your home directory now so you can see how the **rmdir** command works.) If you try to remove the directory **memos** (by issuing the **rmdir** command from your home directory), the command responds as follows:

```
$ rmdir memos<CR>
rmdir:  memos not empty
$
```

To remove the directory **memos**, you must first remove its contents: the subdirectory **tech**, and the files **june.30** and **july.31**. You can remove the **tech** subdirectory by executing the **rmdir** command. For instructions on removing files, see "Accessing and Manipulating Files" later in this chapter.

Once you have removed the contents of the **memos** directory, **memos** itself can be removed. First, however, you must move to its parent directory (your home directory). The **rmdir** command will not work if you are still in the directory you want to remove. From your home directory, type:

rmdir memos<CR>

If **memos** is empty, the command will remove it and return a prompt.

Figure 3-12 summarizes the syntax and capabilities of the **rmdir** command.

Command Recap **rmdir** - remove a directory		
command	*options*	*arguments*
rmdir	none	*directoryname(s)*
Description:	**rmdir** removes specified directories if they do not contain files and/or subdirectories.	
Remarks:	If the directory is empty, it is removed and the system returns a prompt. If the directory contains files or subdirectories, the command returns the message, `rmdir:` *directoryname* `not empty`.	

Figure 3-12: Summary of the **rmdir** Command

Accessing and Manipulating Files

This section introduces several UNIX system commands that access and manipulate files in the file system structure. Information in this section is organized into two parts; basic and advanced. The part devoted to basic commands is fundamental to using the file system; the advanced commands offer more sophisticated information processing techniques for working with files.

Basic Commands

This section discusses UNIX system commands that are necessary for accessing and using the files in the directory structure. Figure 3-13 lists these commands.

Command	Function
cat	prints the contents of a specified file on a terminal
pg	prints the contents of a specified file on a terminal in chunks or pages
pr	prints a partially formatted version of a specified file on the terminal
lp	requests a paper copy of a file from a line printer
cp	makes a duplicate copy of an existing file
mv	moves and renames a file
rm	removes a file
wc	reports the number of lines, words, and characters in a file
chmod	changes permission modes for a file (or a directory)

Figure 3-13: Basic Commands for Using Files

Each command is discussed in detail and summarized in a table that you can easily reference later. These tables will allow you to review the syntax and capabilities of these commands at a glance.

Displaying a File's Contents: the cat, pg, and pr Commands

The UNIX system provides three commands for displaying and printing the contents of a file or files: **cat, pg,** and **pr**. The **cat** command (short for concatenate) outputs the contents of the file(s) specified. This output is displayed on your terminal screen unless you tell **cat** to direct it to another file or a new command.

The **pg** command is particularly useful when you want to read the contents of a long file because it displays the text of a file in pages a screenful at a time. The **pr** command formats specified files and displays them on your terminal or, if you so request, directs the formatted output to a printer (see the **lp** command in this chapter).

The following sections describe how to use the **cat**, **pg**, and **pr** commands.

Concatenate and Print Contents of a File: the cat *Command*

The **cat** command displays the contents of a file or files. For example, say you are located in the directory **letters** (in the sample file system) and you want to display the contents of the file **johnson**. Type the command line shown on the screen and you will receive the following output:

```
$ cat johnson<CR>
March 5, 1986

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.

Dear Mr. Johnson:

I enjoyed speaking with you this morning
about your company's plans to automate
your business.
Enclosed please find
the material you requested
about AB&C's line of computers
and office automation software.

If I can be of further assistance to you,
please don't hesitate to call.

Yours truly,

John Howe
$
```

To display the contents of two (or more) files, simply type the names of the files you want to see on the command line. For example, to display the contents of the files **johnson** and **sanders**, type:

```
$ cat johnson sanders<CR>
```

The **cat** command reads **johnson** and **sanders** and displays their contents in that order on your terminal.

```
$ cat johnson sanders<CR>
March 5, 1986

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.

Dear Mr. Johnson:

I enjoyed speaking with you this morning
.
.

Yours truly,

John Howe

March 5, 1986

Mrs. D.L. Sanders
Sanders Research, Inc.
43 Nassau Street
Princeton, N.J.

Dear Mrs. Sanders:

My colleagues and I have been following, with great interest,
.
.

Sincerely,

John Howe
$
```

To direct the output of the **cat** command to another file or to a new command, see the sections in Chapter 7 that discuss input and output redirection.

Figure 3-14 summarizes the syntax and capabilities of the **cat** command.

Command Recap **cat** - concatenate and print a file's contents		
command	*options*	*arguments*
cat	available*	*filename(s)*
Description:	The **cat** command reads the name of each file specified on the command line and displays its contents.	
Remarks:	If a specified file exists and is readable, its contents are displayed on the terminal screen; otherwise, the message `cat: cannot open` *filename* appears on the screen. To display the contents of a directory, use the **ls** command.	

* See the **cat**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-14: Summary of the **cat** Command

Paging Through the Contents of a File: the **pg** *Command*

The command **pg** (short for page) allows you to examine the contents of a file or files, page by page, on a terminal. The **pg** command displays the text of a file in pages (chunks) followed by a colon prompt (:), a signal that the program is waiting for your instructions. Possible instructions you can then issue include requests for the command to continue displaying the file's contents a page at a time, and a request that the command search through the file(s) to locate a specific character pattern. Figure 3-15 summarizes some of the available instructions.

Command*	Function
h	help; display list of available **pg**† commands
q or **Q**	quit **pg** perusal mode
<CR>	display next page of text
l	display next line of text
d or **^d**	display additional half page of text
. or **^l**	redisplay current page of text
f	skip next page of text and display following one
n	begin displaying next file you specified on command line
p	display previous file specified on command line
$	display last page of text in file currently displayed
/pattern	search forward in file for specified character pattern
?*pattern*	search backward in file for specified character pattern

* Most commands can be typed with a number preceding them. For example, 1 (display next page), −1 (display previous page), or 1 (display first page of text).

† See the *User's Reference Manual* for a detailed explanation of all available **pg** commands.

Figure 3-15: Summary of Commands to Use with **pg**

The **pg** command is useful when you want to read a long file or a series of files because the program pauses after displaying each page, allowing time to examine it. The size of the page displayed depends on the terminal. For example, on a terminal capable of displaying twenty-four lines, one page is defined as twenty-three lines of text and a line containing a colon. However, if a file is less than twenty-three lines long, its page size will be the number of lines in the file plus one (for the colon).

To peruse the contents of a file with **pg**, use the following command line format:

pg *filename(s)* **<CR>**

For example, to display the contents of the file **outline** in the sample file system, type:

pg outline <CR>

The first page of the file will appear on the screen. Because the file has more lines in it than can be displayed on one page, a colon appears at the bottom of the screen. This is a reminder to you that there is more of the file to be seen. When you are ready to read more, press the RETURN key and **pg** will print the next page of the file.

The following screen summarizes our discussion of the **pg** command this far.

```
$ pg outline<CR>
After you analyze the subject for your
report, you must consider organizing and
arranging the material you want to use in
writing it.
      .
      .
      .
An outline is an effective method of
organizing the material.  The outline
is a type of blueprint or skeleton,
a framework for you the builder-writer
of the report; in a sense it is a recipe
:<CR>
```

After you press the RETURN key, **pg** will resume printing the file's contents on the screen.

```
that contains the names of the
ingredients and the order in which
to use them.
     .
     .
     .
Your outline need not be elaborate or
overly detailed; it is simply a guide you
may consult as you write, to be varied,
if need be, when additional important
ideas are suggested in the actual writing.
(EOF):
```

Notice the line at the bottom of the screen containing the string `(EOF):`. This expression `(EOF)` means you have reached the end of the file. The colon prompt is a cue for you to issue another command.

When you have finished examining the file, press the RETURN key; a prompt will appear on your terminal. (Typing **q** or **Q** and pressing the RETURN key also gives you a prompt.) Or you can use one of the other available commands, depending on your needs. In addition, there are a number of options that can be specified on the **pg** command line (see the **pg**(1) page in the *User's Reference Manual*).

Proper execution of the **pg** command depends on specifying the type of terminal you are using. This is because the **pg** program was designed to be flexible enough to run on many different terminals; how it is executed differs from terminal to terminal. By specifying one type, you are telling this command:

- how many lines to print
- how many columns to print
- how to clear the screen
- how to highlight prompt signs or other words
- how to erase the current line

To specify a terminal type, assign the code for your terminal to the TERM variable in your **.profile** file. (For more information about TERM and **.profile**, see Chapter 7; for instructions on setting the TERM variable, see Appendix F.)

Figure 3-16 summarizes the syntax and capabilities of the **pg** command.

Command Recap **pg** - display a file's contents in chunks or pages		
command	*options*	*arguments*
pg	available*	*filename(s)*
Description:	The **pg** command displays the contents of the specified file(s) in pages.	
Remarks:	After displaying a page of text, the **pg** command awaits instructions from you to do one of the following: continue to display text, search for a pattern of characters, or exit the **pg** perusal mode. In addition, a number of options are available. For example, you can display a section of a file beginning at a specific line or at a line containing a certain sequence or pattern. You can also opt to go back and review text that has already been displayed.	

* See the **pg**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-16: Summary of the **pg** Command

Print Partially Formatted Contents of a File: the **pr** *Command*

The **pr** command is used to prepare files for printing. It supplies titles and headings, paginates, and prints a file, in any of various page lengths and widths, on your terminal screen.

You have the option of requesting that the command print its output on another device, such as a line printer (read the discussion of the **lp** command in this section). You can also direct the output of **pr** to a different file (see the sections on input and output redirection in Chapter 7).

If you choose not to specify any of the available options, the **pr** command produces output in a single column that contains sixty-six lines per page and is preceded by a short heading. The heading consists of five lines: two blank lines; a line containing the date, time, file name, and page number; and two more blank lines. The formatted file is followed by five blank lines. (Complete sets of text formatting tools are available on UNIX systems equipped with the Documenter's Workbench Software. Check with your system administrator to see if this software is available to you.)

The **pr** command is often used together with the **lp** command to provide a paper copy of text as it was entered into a file. (See the section on the **lp** command for details.) However, you can also use the **pr** command to format and print the contents of a file on your terminal. For example, to review the contents of the file **johnson** in the sample file system, type:

pr johnson <CR>

The following screen gives an example of output from this command.

```
$ pr johnson<CR>


Mar  5 15:43 1986 johnson Page 1


March 5, 1986

Mr. Ron Johnson
Layton Printing
52 Hudson Street
New York, N.Y.


Dear Mr. Johnson:

I enjoyed speaking with you this morning
about your company's plans to automate
your business.
Enclosed please find
the material you requested
about AB&C's line of computers
and office automation software.

If I can be of further assistance to you,
please don't hesitate to call.

Yours truly,

John Howe
.
.
$
```

The ellipses after the last line in the file represent the remaining lines (all blank in this case) that **pr** formatted into the output (so that each page contains a total of sixty-six lines). If you are working on a video display terminal, which allows you to view twenty-four lines at a time, the entire sixty-six lines of the formatted file will be printed rapidly without pause. This means that the

first forty-two lines will roll off the top of your screen, making it impossible for you to read them unless you have the ability to roll back a screen or two. However, if the file you are examining is particularly long, even this ability may not be sufficient to allow you to read the file.

In such cases, type <^s> (control-s) to interrupt the flow of printing on your screen. When you are ready to continue, type <^q> (control-q) to resume printing.

Figure 3-17 summarizes the syntax and capabilities of the **pr** command.

Command Recap **pr** - print formatted contents of a file		
command	*options*	*arguments*
pr	available*	*filename(s)*
Description:	The **pr** command produces a formatted copy of a file(s) on your terminal screen unless you specify otherwise. It prints the text of the file(s) on sixty-six line pages, and places five blank lines at the bottom of each page and a five-line heading at the top of each page. The heading includes: two blank lines; a line containing the date, time, file name, and page number; and two additional blank lines.	
Remarks:	If a specified file exists, its contents are formatted and displaye; if not, the message `pr: can't open` *filename* is printed. The **pr** command is often used with the **lp** command to produce a paper copy of a file. It can also be used to review a file on a video display terminal. To stop and restart the printing of a file on a terminal, type <^**s**> and <^**q**>, respectively.	

* See the **pr**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-17: Summary of the **pr** Command

Requesting a Paper Copy of a File: the lp Command

Some terminals have built-in printers that allow you to get paper copies of files. If you have such a terminal, you can get a paper copy of your file simply by turning on the printer and executing the **cat** or **pr** command. However, if you are using a video display terminal, you must send a request for a paper copy of a file to a printer (see Figure 3-18). The command **lp** (short for line printer) allows you to do this.

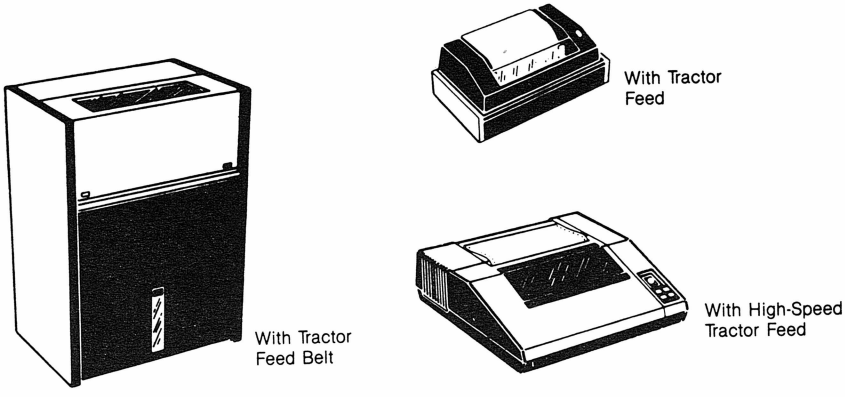


Figure 3-18: Examples of Teletype Model 40 Line Printers

To execute **lp**, follow this format:

lp *filename* <**CR**>

For example, to print the file **johnson** on a line printer, type the following command line:

lp johnson<CR>

The system responds with the name (or type) of the printer on which the file will be printed, and an identification (ID) number for your request.

```
$ lp johnson<CR>
request id is laser-6885  (1 file)
$
```

The system response shows that your job is to be printed on a laser printer (this system's default type of printer), has a request ID number of 6885, and includes one file.

The **-d***dest* (short for destination) option on the command line causes your file to be printed on another available device that you specify in the *dest* argument. The **-m** option causes mail to be sent to you stating the job has been completed.

To cancel a request to a printer, type the command **cancel** and specify the request ID number. For example, to cancel your request for a printing of the file **letters** (request ID laser-6885), type:

cancel laser-6885<CR>

To check the status of a line printer job that it is in progress, or to get its request ID number, execute the **lpstat** command. This command also provides a complete listing of every printer available on your system. Which printers are available to you depends on your UNIX system facility. Ask your system administrator for the names of available line printers, or type the following command line:

lpstat -v<CR>

Figure 3-19 summarizes the syntax and capabilities of the **lp** command.

Command Recap

lp - request paper copy of file from a line printer

command	*options*	*arguments*
lp	**–d, –m,** and others*	*file(s)*

Description: The **lp** command requests that specified files be printed by a line printer, thus providing paper copies of the contents.

Options:

–d*dest* Allows you to choose *dest* as the printer or type of printer to produce the paper copy. If you do not use this option, the **lp** program specifies the printer for you.

–m Sends a message to you via **mail** after the printing is complete.

Remarks: You can cancel a request to the line printer by typing **cancel** and the request **ID** furnished to you by the system when the request was acknowledged.

Check with your system administrator for information on additional and/or different commands for printers that may be available at your location.

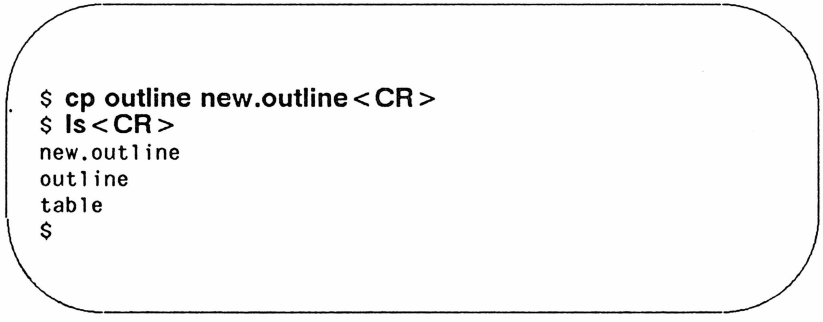
* See the **lp**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-19: Summary of the **lp** Command

Making a Duplicate Copy of a File: the cp Command

When using the UNIX system, you may want to make a copy of a file. For example, you might want to revise a file while leaving the original version intact. The command **cp** (short for copy) copies the complete contents of one file into another. The **cp** command also allows you to copy one or more files from one directory into another while leaving the original file or files in place.

To copy the file named **outline** to a file named **new.outline** in the sample directory, simply type **cp outline new.outline** and press the RETURN key. The system returns the prompt when the copy is made. To verify the existence of the new file, you can type **ls** and press the RETURN key. This command lists the names of all files and directories in the current directory, in this case **draft**. The following screen summarizes these activities.



The UNIX system does not allow you to have two files with the same name in a directory. In this case, because there was no file called **new.outline** when the **cp** command was issued, the system created a new file with that name. However, if a file called **new.outline** had already existed, it would have been replaced by a copy of the file **outline**; the previous version of **new.outline** would have been deleted.

If you had tried to copy the file **outline** to another file named **outline** in the same directory, the system would have told you the file names were identical and returned the prompt to you. If you had then listed the contents of the directory to determine exactly

how many copies of **outline** existed, you would have received the following output on your screen:

```
$ cp outline outline<CR>
cp: outline and outline are identical
$ ls<CR>
outline
table
$
```

The UNIX system does allow you to have two files with the same name as long as they are in different directories. For example, the system would let you copy the file **outline** from the **draft** directory to another file named **outline** in the **letters** directory. If you were in the **draft** directory, you could use any one of four command lines. In the first two command lines, you specify the name of the new file you are creating by making a copy.

- **cp outline /user1/starship/letters/outline<CR>** (full path name specified)
- **cp outline ../letters/outline<CR>** (relative path name specified)

However, the **cp** command does not require that you specify the name of the new file. If you do not include a name for it on the command line, **cp** gives your new file the same name as the original one, by default. Therefore you could also use either of these command lines:

- **cp outline /user1/starship/letters<CR>** (full path name specified)
- **cp outline ../letters<CR>** (relative path name specified)

In any of these four cases, **cp** will make a copy of the **outline** file in the **letters** directory and call it **outline**, too.

Of course, if you want to give your new file a different name, you must specify it. For example, to copy the file **outline** in the **draft** directory to a file named **outline.vers2** in the **letters** directory, you can use either of the following command lines:

- **cp outline /user1/starship/letters/outline.vers2 <CR>** (full path name)
- **cp outline ../letters/outline.vers2 <CR>** (relative path name)

When assigning new names, keep in mind the conventions for naming directories and files described in "Naming Directories and Files" in this chapter.

Figure 3-20 summarizes the syntax and capabilities of the **cp** command.

Command Recap **cp** - make a copy of a file		
command	*options*	*arguments*
cp	none	*file1 file2* *file(s) directory*
Description:	**cp** allows you to make a copy of *file1* and call it *file2* leaving *file1* intact or to copy one or more files into a different directory.	
Remarks:	When you are copying *file1* to *file2* and a file called *file2* already exists, the **cp** command overwrites the first version of *file2* with a copy of *file1* and calls it *file2*. The first version of *file2* is deleted. You cannot copy directories with the **cp** command.	

Figure 3-20: Summary of the **cp** Command

Moving and Renaming a File: the mv Command

The command **mv** (short for move) allows you to rename a file in the same directory or to move a file from one directory to another. If you move a file to a different directory, the file can be renamed or it can retain its original name.

To rename a file within one directory, follow this format:

mv *file1 file2*<**CR**>

The **mv** command changes a file's name from *file1* to *file2* and deletes *file1*. Remember that the names *file1* and *file2* can be any valid names, including path names.

For example, if you are in the directory **draft** in the sample file system and you would like to rename the file **table** to **new.table**, simply type **mv table new.table** and press the RETURN key. If the command executes successfully, you will receive a prompt. To verify that the file **new.table** exists, you can list the contents of the directory by typing **ls** and pressing the RETURN key. The screen shows your input and the system's output as follows:

```
$ mv table new.table<CR>
$ ls<CR>
new.table
outline
$
```

You can also move a file from one directory to another, keeping the same name or changing it to a different one. To move the file without changing its name, use the following command line:

mv *file(s) directory*<**CR**>

The file and directory names can be any valid names, including path names.

For example, say you want to move the file **table** from the current directory named **draft** (whose full path name is **/user1/starship/draft**) to a file with the same name in the directory **letters** (whose relative path name from **draft** is **../letters** and whose full path name is **/user1/starship/letters**), you can use any one of several command lines, including the following:

```
mv table /user1/starship/letters <CR>

mv table /user1/starship/letters/table <CR>

mv table ../letters <CR>

mv table ../letters/table <CR>

mv /user1/starship/draft/table
    /user1/starship/letters/table <CR>
```

Now suppose you want to rename the file **table** as **table2** when moving it to the directory **letters**. Use any of these command lines:

```
mv table /user1/starship/letters/table2 <CR>

mv table ../letters/table2 <CR>

mv /user1/starship/draft/table2
    /user1/starship/letters/table2 <CR>
```

You can verify that the command worked by using the **ls** command to list the contents of the directory.

Figure 3-21 summarizes the syntax and capabilities of the **mv** command.

Command Recap **mv** - move or rename files		
command	*options*	*arguments*
mv	none	*file1 file2* *file(s) directory*
Description:	**mv** allows you to change the name of a file or to move a file(s) into another directory.	
Remarks:	When you are moving *file1* to *file2*, if a file called *file2* already exists, the **mv** command overwrites the first version of *file2* with *file1* and renames it *file2*. The first version of *file2* is deleted.	

Figure 3-21: Summary of the **mv** Command

Removing a File: the rm Command

When you no longer need a file, you can remove it from your directory by executing the command **rm** (short for remove). The basic format for this command is:

rm *file(s)* <**CR**>

You can remove more than one file at a time by specifying those files you want to delete on the command line with a space separating each filename:

rm *file1 file2 file3* <**CR**>

The system does not save a copy of a file it removes; once you have executed this command, your file is removed permanently.

After you have issued the **rm** command, you can verify its successful execution by running the **ls** command. Since **ls** lists the files in your directory, you'll immediately be able to see whether or not **rm** has executed successfully.

For example, say you have a directory that contains two files, **outline** and **table**. You can remove both files by issuing the **rm** command once. If **rm** is executed successfully, your directory will be empty. Verify this by running the **ls** command.

```
$ rm outline table <CR>
$ ls
$
```

The prompt shows that **outline** and **table** were removed.

Figure 3-22 summarizes the syntax and capabilities of the **rm** command.

Command Recap **rm** - remove a file		
command	*options*	*arguments*
rm	available*	*file(s)*
Description:	**rm** allows you to remove one or more files.	
Remarks:	Files specified as arguments to the **rm** command are removed permanently.	

* See the **rm**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-22: Summary of the **rm** Command

Counting Lines, Words, and Characters in a File: the wc Command

The command **wc** (short for word count) reports the number of lines, words, and characters there are in the file(s) named on the command line. If you name more than one file, the **wc** program counts the number of lines, words, and characters in each specified file and then totals the counts. In addition, you can direct the **wc** program to give you only a line, a word, or a character count by using the **–l**, **–w**, or **–c** options, respectively.

To determine the number of lines, words, and characters in a file, use the following format on the command line:

wc file1 <CR>

The system responds with a line in the following format:

l w c file1

where

- *l* represents the number of lines in *file1*
- *w* represents the number of words in *file1*
- *c* represents the number of characters in *file1*

For example, to count the lines, words, and characters in the file **johnson** (located in the current directory, **letters**), type the following command line:

```
$ wc johnson <CR>
24      66      406 johnson
$
```

The system response means that the file **johnson** has twenty-four lines, sixty-six words, and 406 characters.

To count the lines, words, and characters in more than one file, use this format:

wc *file1 file2* **<CR>**

The system responds in the following format:

```
l     w     c     file1
l     w     c     file2
l     w     c     total
```

Line, word, and character counts for *file1* and *file2* are displayed on separate lines and the combined counts appear on the last line beside the word `total`.

For example, ask the **wc** program to count the lines, words, and characters in the files **johnson** and **sanders** in the current directory.

```
$ wc johnson sanders<CR>
    24      66      406 johnson
    28      92      559 sanders
    52     158      965 total
$
```

The first line reports that the **johnson** file has twenty-four lines, sixty-six words, and 406 characters. The second line reports twenty-eight lines, ninety-two words, and 559 characters in the **sanders** file. The last line shows that these two files together have a total of fifty-two lines, 158 words, and 965 characters.

To get only a line, a word, or a character count, select the appropriate command line format from the following lines:

wc **-l** *file1*<**CR**> (line count)
wc **-w** *file1*<**CR**> (word count)
wc **-c** *file1*<**CR**> (character count)

For example, if you use the **–l** option, the system reports only the number of lines in **sanders**.

```
$ wc –l sanders<CR>
      28 sanders
$
```

If the **–w** or **–c** option had been specified instead, the command would have reported the number of words or characters, respectively, in the file.

Figure 3-23 summarizes the syntax and capabilities of the **wc** command.

Command Recap **wc** - count lines, words, and characters in a file		
command	*options*	*arguments*
wc	**–l, –w, –c**	*file(s)*
Description:	**wc** counts lines, words, and characters in the specified file(s), keeping a total count of all tallies when more than one file is specified.	
Options	**–l** counts the number of lines in the specified file(s) **–w** counts the number of words in the specified file(s) **–c** counts the number of characters in the specified file(s)	
Remarks:	When a file name is specified in the command line, it is printed with the count(s) requested.	

Figure 3-23: Summary of the **wc** Command

Protecting Your Files: the chmod Command

The command **chmod** (short for change mode) allows you to decide who can read, write, and use your files and who cannot. Because the UNIX operating system is a multi-user system, you usually do not work alone in the file system. System users can follow path names to various directories and read and use files belonging to one another, as long as they have permission to do so.

If you own a file, you can decide who has the right to read it, write in it (make changes to it), or, if it is a program, to execute it. You can also restrict permissions for directories with the **chmod** command. When you grant execute permission for a directory, you allow the specified users to **cd** to it and list its contents with the **ls** command.

To assign these types of permissions, use the following three symbols:

- **r** allows system users to read a file or to copy its contents
- **w** allows system users to write changes into a file (or a copy of it)
- **x** allows system users to run an executable file

To specify the users to whom you are granting (or denying) these types of permission, use these three symbols:

- **u** you, the owner of your files and directories (**u** is short for user)
- **g** members of the group to which you belong (the group could consist of team members working on a project, members of a department, or a group arbitrarily designated by the person who set up your UNIX system account)
- **o** all other system users

When you create a file or a directory, the system automatically grants or denies permission to you, members of your group, and other system users. You can alter this automatic action by modifying your environment (see Chapter 7 for details). Moreover, regardless of how the permissions are granted when a file is created, as the owner of the file or directory you always have the option of changing them. For example, you may want to keep certain files private and reserve them for your exclusive use. You may want to grant permission to read and write changes into a file to members of your group and all other system users as well. Or you may share a program with members of your group by granting them permission to execute it.

How to Determine Existing Permissions

You can determine what permissions are currently in effect on a file or a directory by using the command that produces a long listing of a directory's contents: **ls −l**. For example, typing **ls −l** and pressing the RETURN key while in the directory named **starship/bin** in the sample file system produces the following output:

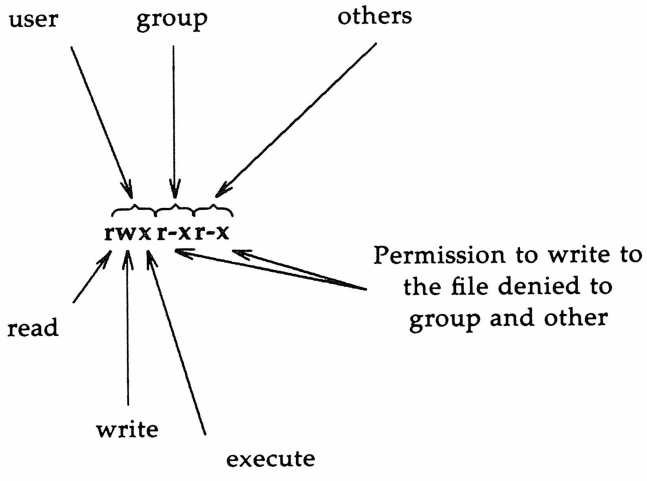
```
$ ls −l<CR>
total 35
-rwxr-xr-x  1 starship  project  9346  Nov 1  08:06  display
-rw-r--r--  1 starship  project  6428  Dec 2  10:24  list
drwx--x--x  2 starship  project    32  Nov 8  15:32  tools
$
```

Permissions for the **display** and **list** files and the **tools** directory are shown on the left of the screen under the line `total 35`, and appear in this format:

`-rwxr-xr-x` (for the **display** file)
`-rw-r--r--` (for the **list** file))
`drwx--x---` (for the **tools** directory)

After the initial character, which describes the file type (for example,
a - (dash) symbolizes a regular file and a CWd a directory), the other nine characters that set the permissions comprise three sets of three characters. The first set refers to permissions for the owner, the second set to permissions for group members, and the last set to permissions for all other system users. Within each set of characters, the `r`, `w`, and `x` show the permissions currently granted to each category. If a dash appears instead of an `r`, `w`, or `x`, permission to read, write, or execute is denied.

The following diagram summarizes this breakdown for the file named **display**.



As you can see, the owner has `r`, `w`, and `x` permissions and members of the group and other system users have `r` and `x` permissions.

There are two exceptions to this notation system. Occasionally the letter `s` or the letter `l` may appear in the permissions line, instead of an `r`, `w` or `x`. The letter `s` (short for set user ID or set group ID) represents a special type of permission to execute a file. It appears where you normally see an `x` (or -) for the user or

group (the first and second sets of permissions). From a user's point of view it is equivalent to an x in the same position; it implies that execute permission exists. It is significant only for programmers and system administrators. (See the *Administration Guide* for details about setting the user or group ID.)

The letter l is the symbol for lock enabling. It does not mean that the file has been locked. It simply means that the function of locking is enabled, or possible, for this file. The file may or may not be locked; that cannot be determined by the presence or absence of the letter l.

How to Change Existing Permissions

After you have determined what permissions are in effect, you can change them by executing the **chmod** command in the following format:

chmod *whopermission file(s)* <**CR**>

The following list defines each component of this command line.

chmod	name of the program
who	one of three user groups (**u, g,** or **o**) **u** user **g** group **o** others
+ or **−**	instruction that grants (**+**) or denies (**−**) permission
permission	any combination of three authorizations (**r, w,** and **x**) **r** read **w** write **x** execute
file(s)	file (or directory) name(s) listed; assumed to be branches from your current directory, unless you use full pathnames.

NOTE: The **chmod** command will not work if you type a space(s) between *who*, the instruction that gives () or denies (−) permission, and the *permission*.

The following examples show a few possible ways to use the **chmod** command. As the owner of **display**, you can read, write, and run this executable file. You can protect the file against being accidentally changed by denying yourself write (w) permission. To do this, type the command line:

chmod u − w display < CR >

After receiving the prompt, type **ls −l** and press the RETURN key to verify that this permission has been changed, as shown in the following screen.

```
$ chmod u-w display<CR>
$ ls -l<CR>
total 35
-r-xr-xr-x  1 starship  project  9346  Nov 1  08:06  display
rw-r--r--   1 starship  project  6428  Dec 2  10:24  list
drwx--x--x  2 starship  project    32  Nov 8  15:32  tools
$
```

As you can see, you no longer have permission to write changes into the file. You will not be able to change this file until you restore write permission for yourself.

Now consider another example. Notice that permission to write into the file **display** has been denied to members of your group and other system users. However, they do have read permission. This means they can copy the file into their own directories and then make changes to it. To prevent all system users from copying this file, you can deny them read permission by typing:

chmod go – r display <CR>

The **g** and **o** stand for group members and all other system users, respectively, and the **– r** denies them permission to read or copy the file. Check the results with the **ls – l** command.

```
$ chmod go – r display <CR>
$ ls – l <CR>
total 35
-rwx--x--x  1 starship  project  9346  Nov 1  08:06  display
rw-r--r--   1 starship  project  6428  Dec 2  10:24  list
drwx--x--x  2 starship  project    32  Nov 8  15:32  tools
$
```

A Note on Permissions and Directories

You can use the **chmod** command to grant or deny permission for directories as well as files. Simply specify a directory name instead of a file name on the command line.

However, consider the impact on various system users of changing permissions for directories. For example, say you grant read permission for a directory to yourself (**u**), members of your group (**g**), and other system users (**o**). Every user who has access to the system will be able to read the names of the files contained in that directory by running the **ls – l** command. Similarly, granting write permission allows the designated users to create new files in the directory and remove existing ones. Granting permission to execute the directory allows designated users to move to that directory (and make it their current directory) by using the **cd** command.

An Alternative Method

There are two methods by which the **chmod** command can be executed. The method described above, in which symbols such as **r**, **w**, and **x** are used to specify permissions, is called the symbolic method.

An alternative method is the octal method. Its format requires you to specify permissions using three octal numbers, ranging from 0 to 7. (The octal number system is different from the decimal system that we typically use on a day-to-day basis.) To learn how to use the octal method, see the **chmod**(1) page in the *User's Reference Manual*.

Figure 3-24 summarizes the syntax and capabilities of the **chmod** command.

Command Recap **chmod** - change permission modes for files (and directories)		
command	*instruction*	*arguments*
chmod	**who – permission**	*filename(s)* *directoryname(s)*
Description:	**chmod** gives () or removes (–) permission to read, write, and execute files for three categories of system users: **user** (you), **group** (members of your group), and **other** (all other users able to access the system on which you are working).	
Remarks:	The instruction set can be represented in either octal or symbolic terms.	

Figure 3-24: Summary of the **chmod** Command

Advanced Commands

Use of the commands already introduced will increase your familiarity with the file system. As this familiarity increases, so might your need for more sophisticated information processing techniques when working with files. This section introduces three commands that provide just that.

diff finds differences between two files

grep searches for a pattern in a file

sort sorts and merges files

For additional information about these commands refer to the *User's Reference Manual.*

Identifying Differences Between Files: the diff Command

The **diff** command locates and reports all differences between two files and tells you how to change the first file so that it is a duplicate of the second. The basic format for the command is:

diff *file1 file2* **<CR>**

If *file1* and *file2* are identical, the system returns a prompt to you. If they are not, the **diff** command instructs you on how to change the first file so it matches the second by using **ed** (line editor) commands. (See Chapter 5 for details about the line editor.) The UNIX system flags lines in *file1* (to be changed) with the < (less than) symbol, and lines in *file2* (the model text) with the > (greater than) symbol.

For example, say you execute the **diff** command to identify the differences between the files **johnson** and **mcdonough**. The **mcdonough** file contains the same letter that is in the **johnson** file, with appropriate changes for a different recipient. The **diff** command will identify those changes as follows:

```
3,6c3,6
< Mr. Ron Johnson
< Layton Printing
< 52 Hudson Street
< New York, N.Y.
---
> Mr. J.J. McDonough
> Ubu Press
> 37 Chico Place
> Springfield, N.J.
9c9
< Dear Mr. Johnson:
---
> Dear Mr. McDonough:
```

The first line of output from **diff** is :

```
3,6c3,6
```

This means that if you want **johnson** to match **mcdonough**, you must change (c) lines 3 through 6 in **johnson** to lines 3 through 6 in **mcdonough**. The **diff** command then displays both sets of lines.

If you make these changes (using a text editor such as **ed** or **vi**), the **johnson** file will be identical to the **sanders** file. Remember, the **diff** command identifies differences between specified files. If you want to make an identical copy of a file, use the **cp** command.

Figure 3-25 summarizes the syntax and capabilities of the **diff** command.

Command Recap **diff** - finds differences between two files		
command	*options*	*arguments*
diff	available*	*file1 file2*
Description:	The **diff** command reports what lines are different in two files and what you must do to make the first file identical to the second.	
Remarks:	Instructions on how to change a file to bring it into agreement with another file are line editor (**ed**) commands: **a** (append), **c** (change), and **d** (delete). Numbers given with **a**, **c**, or **d** show the lines to be modified. Also used are the symbols < (showing a line from the first file) and > (showing a line from the second file).	

* See the **diff**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-25: Summary of the **diff** Command

Searching a File for a Pattern: the grep Command

You can instruct the UNIX system to search through a file for a specific word, phrase, or group of characters by executing the command **grep** (short for **g**lobally search for a **r**egular **e**xpression and **p**rint). Put simply, a regular expression is any pattern of characters (be it a word, a phrase, or an equation) that you specify.

The basic format for the command line is:

grep *pattern file(s)* **<CR>**

For example, to locate any lines that contain the word automation in the file **johnson**, type:

grep automation johnson <CR>

The system responds:

```
$ grep automation johnson <CR>
and office automation software.
$
```

The output consists of all the lines in the file **johnson** that contain the pattern for which you were searching (automation).

If the pattern contains multiple words or any character that conveys special meaning to the UNIX system, (such as **$**, **¦**, *****, **?**, and so on), the entire pattern must be enclosed in single quotes. (For an explanation of the special meaning for these and other characters see "Metacharacters" in Chapter 7.) For example, say you want to locate the lines containing the pattern `office automation`. Your command line and the system's response will read:

```
$ grep 'office automation' johnson <CR>
and office automation software.
$
```

But what if you cannot recall which letter contained a reference to office automation; your letter to Mr. Johnson or the one to Mrs. Sanders? Type the following command line to find out:

```
$ grep office automation' johnson sanders <CR>
johnson:and office automation software.
$
```

The output tells you that the pattern `office automation` is found once in the **johnson** file.

In addition to the **grep** command, the UNIX system provides variations of it called **egrep** and **fgrep**, along with several options that enhance the searching powers of the command. See the **grep**(1), **egrep**(1), and **fgrep**(1) pages in the *User's Reference Manual* for further information about these commands.

Figure 3-26 summarizes the syntax and capabilities of the **grep** command.

Command Recap **grep** - searches a file for a pattern		
command	*options*	*arguments*
grep	available*	*pattern file(s)*
Description:	The **grep** command searches through specified file(s) for lines containing a pattern and then prints the lines on which it finds the pattern. If you specify more than one file, the name of the file in which the pattern is found is also reported.	
Remarks:	If the pattern you give contains multiple words or special characters, enclose the pattern in single quotes on the command line.	

* See the **grep**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-26: Summary of the **grep** Command

Sorting and Merging Files: the sort Command

The UNIX system provides an efficient tool called **sort** for sorting and merging files. The format for the command line is:

sort *file(s)* <**CR**>

This command causes lines in the specified files to be sorted and merged in the following order.

- Lines beginning with numbers are sorted by digit and listed before lines beginning with letters.
- Lines beginning with upper case letters are listed before lines beginning with lower case letters.
- Lines beginning with symbols such as *, %, or @, are sorted on the basis of the symbol's ASCII representation.

For example, let's say you have two files, **group1** and **group2**, each containing a list of names. You want to sort each list alphabetically and then interleave the two lists into one. First, display the contents of the files by executing the **cat** command on each.

```
$ cat group1 <CR>
Smith, Allyn
Jones, Barbara
Cook, Karen
Moore, Peter
Wolf, Robert
$ cat group2 <CR>
Frank, M. Jay
Nelson, James
West, Donna
Hill, Charles
Morgan, Kristine
$
```

(Instead of printing these two files individually, you could have requested both files on the same command line. If you had typed **cat group1 group2** and pressed the RETURN key, the output would have been the same.)

Now sort and merge the contents of the two files by executing the **sort** command. The output of the **sort** program will be printed on the terminal screen unless you specify otherwise.

```
$ sort group1 group2 <CR>
Cook, Karen
Frank, M. Jay
Hill, Charles
Jones, Barbara
Moore, Peter
Morgan, Kristine
Nelson, James
Smith, Allyn
West, Donna
Wolf, Robert
$
```

In addition to combining simple lists as in the example, the **sort** command can rearrange lines and parts of lines (called fields) according to a number of other specifications you designate on the command line. The possible specifications are complex and beyond the scope of this text. Refer to the *User's Reference Manual* for a full description of available options.

Figure 3-27 summarizes the syntax and capabilities of the **sort** command.

Command Recap **sort** - sorts and merges files		
command	*options*	*arguments*
sort	available*	*file(s)*
Description:	The **sort** command sorts and merges lines from a file or files you specify and displays its output on your terminal screen.	
Remarks:	If no options are specified on the command line, lines are sorted and merged in the order defined by the ASCII representations of the characters in the lines.	

* See the **sort**(1) page in the *User's Reference Manual* for all available options and an explanation of their capabilities.

Figure 3-27: Summary of the **sort** Command

Summary

This chapter described the structure of the file system and presented ways to use and to navigate through the file system by using UNIX system commands. The next chapter gives you an overview of a variety of UNIX system capabilities: text editing, using the shell as a command language, communicating electronically with other system users, and programming and developing software.

[This page left blank.]

PART 2: UNIX SYSTEM TUTORIALS

Chapter 4: Overview of the Tutorials

Introduction 4-1

Text Editing 4-2
- What is a Text Editor? 4-2
- How Does a Text Editor Work? 4-2
  - Text Editing Buffers 4-3
  - Modes of Operation 4-3
- Line Editor 4-4
- Screen Editor 4-5

The Shell 4-7
- Customizing Your Computing Environment 4-8
- Programming in the Shell 4-9

Communicating Electronically 4-12

Programming in the System 4-13

Introduction

This chapter serves as a transition between the overview that comprises the first three chapters and the tutorials in the following four chapters. Specifically, it provides an overview of the subjects covered in these tutorials: text editing, working in the shell, and communicating electronically. Text editing is covered in Chapter 5, "Line Editor Tutorial," and Chapter 6, "Screen Editor Tutorial." How to work and program in the shell is taught in Chapter 7, "Shell Tutorial," and methods of electronic communication are covered in Chapter 8, "Communication Tutorial."

Text Editing

Using the file system is a way of life in a UNIX system environment. This section will teach you how to create and modify files with a software tool called a text editor. The section begins by explaining what a text editor is and how it works. Then it introduces two types of text editors supported on the UNIX system: the line editor, **ed**, and the screen editor, **vi** (short for visual editor). A comparison of the two editors is also included. For detailed information about **ed** and **vi**, see Chapters 5 and 6.

What is a Text Editor?

Whenever you revise a letter, memo, or report, you must perform one or more of the following tasks: insert new or additional material, delete unneeded material, transpose material (sometimes called cutting and pasting), and, finally, prepare a clean, corrected copy. Text editors perform these tasks at your direction, making writing and revising text much easier and quicker than if done by hand.

The UNIX system text editors, like the UNIX system shell, are interactive programs; they accept your commands and then perform the requested functions. From the shell's point of view, the editors are executable programs.

A major difference between a text editor and the shell, however, is the set of commands that each recognizes. All the commands introduced up to this point belong to the shell's command set. A text editor has its own distinct set of commands that allow you to create, move, add, and delete text in files, as well as acquire text from other files.

How Does a Text Editor Work?

To understand how a text editor works, you need to understand the environment created when you use an editing program and the modes of operation understood by a text editor.

Text Editing Buffers

When you use a text editor to create a new file or modify an existing one, you first ask the shell to put the editor in control of your computing session. As soon as the editor takes over, it allocates a temporary work space called the editing buffer; any information that you enter while editing a file is stored in this buffer where you can modify it.

Because the buffer is a temporary work space, any text you enter and any changes you make to it are also temporary. The buffer and its contents will exist only as long as you are editing. If you want to save the file, you must tell the text editor to write the contents of the buffer into a file. The file is then stored in the computer's memory. If you do not, the buffer's contents will disappear when you leave the editing program. To prevent this from happening, the text editors send you a reminder to write your file if you attempt to end an editing session without doing so.

NOTE: If you have made a critical mistake or are unhappy with the edited version, you can choose to leave the editor without writing the file. By doing so, you leave the original file intact; the edited copy disappears.

Regardless of whether you are creating a new file or updating an existing one, the text in the buffer is organized into lines. A line of text is simply a series of characters that appears horizontally across the screen and is ended when you press the RETURN key. Occasionally, files may contain a line of text that is too long to fit on the terminal screen. Some terminals automatically display the continuation of the line on the next row of the screen; others do not.

Modes of Operation

Text editors are capable of understanding two modes of operation: command mode and text input mode. When you begin an editing session, you will be placed automatically in command mode. In this mode you can move around in a file, search

for patterns in it, or change existing text. However, you cannot create text while you are in command mode. To do this you must be in text input mode. While you are in this mode, any characters you type are placed in the buffer as part of your text file. When you have finished entering text and want to run editing commands again, you must return to command mode.

Because a typical editing session involves moving back and forth between these two modes, you may sometimes forget which mode you are working in. You may try to enter text while in command mode or to enter a command while in input mode. This is something even experienced users do from time to time. It will not take long to recognize your mistake and determine the solution after you complete the tutorials in Chapters 5 and 6.

Line Editor

The line editor, accessed by the **ed** command, is a fast, versatile program for preparing text files. It is called a line editor because it manipulates text on a line-by-line basis. This means you must specify, by line number, the line containing the text you want to change. Then **ed** prints the line on the screen where you can modify it.

This text editor provides commands with which you can change lines, print lines, read and write files, and enter text. In addition, you can invoke the line editor from a shell program; something you cannot do with the screen editor. (See Chapter 7 for information on basic shell programming techniques.)

The line editor (**ed**) works well on video display terminals and paper printing terminals. It will also accommodate you if you are using a slow-speed telephone line. (The visual editor, **vi**, can be used only on video display terminals.) Refer to Chapter 5, "Line Editor Tutorial," for instructions on how to use this editing tool. Also see Appendix C for a summary of line editor commands.

Screen Editor

The screen editor, accessed by the **vi** command, is a display-oriented, interactive software tool. It allows you to view the file you are editing a page at a time. This editor works most efficiently when used on a video display terminal operating at 1200 or higher baud.

For the most part, you modify a file (by adding, deleting, or changing text) by positioning the cursor at the point on the screen where the modification is to be made and then making the change. The screen editor immediately displays the results of your editing; you can see the change you made in the context of the surrounding text. Because of this feature, the screen editor is considered more sophisticated than the line editor.

Furthermore, the screen editor offers a choice of commands. For example, a number of screen editor commands allow you to move the cursor around a file. Other commands scroll the file up or down on the screen. Still other commands allow you to change existing text or to create new text. In addition to its own set of commands, the screen editor can access line editor commands.

The trade-off for the screen editor's speed, visual appeal, efficiency, and power is the heavy demand it places on the computer's processing time. Every time you make a change, no matter how simple, **vi** must update the screen. Refer to Chapter 6, "Screen Editor Tutorial," for instructions on how to use this editor. Appendix D contains a summary of screen editor commands, and Figure 4-1 compares the features of the line editor (**ed**) and the screen editor (**vi**).

Feature	Line Editor (ed)	Screen Editor (vi)
Recommended terminal type	Video display or paper-printing	Video display
Speed	Accommodates high- and low-speed data transmission lines.	Works best via high-speed data transmission lines (1200+ baud).
Versatility	Can be specified to run from shell scripts as well as used during editing sessions.	Must be used interactively during editing sessions.
Sophistication	Changes text quickly. Uses comparatively small amounts of processing time.	Changes text easily. However, can make heavy demands on computer resources.
Power	Provides a full set of editing commands. Standard UNIX system text editor.	Provides its own editing commands and recognizes line editor commands as well.
Advantages	There are fewer commands you must learn to use **ed**.	**vi** allows you to see the effects of your editing in the context of a page of text, immediately. (When you use the **ed** editor, making changes and viewing the results are separate steps.)

Figure 4-1: Comparison of Line and Screen Editors (**ed** and **vi**)

The Shell

Every time you log in to the UNIX system you start communicating with the shell, and continue to do so until you log off the system. However, while you are using a text editor, your interaction with the shell is suspended; it resumes as soon as you stop using the editor.

The shell is much like other programs, except that instead of performing one job, as **cat** or **ls** does, it is central to your interactions with the UNIX system. The shell's primary function is to act as a command interpreter between you and the computer system. As an interpreter, the shell translates your requests into language the computer understands, calls requested programs into memory, and executes them.

This section introduces methods of using the shell that enhance your ability to use system features. In addition to using it to run a single program, you may also use the shell to:

- interpret the name of a file or a directory you enter in an abbreviated way using a type of shell shorthand
- redirect the flow of input and output of the programs you run
- execute multiple programs simultaneously or in a pipeline format
- tailor your computing environment to meet your individual needs

In addition to being the command language interpreter, the shell is a programming language. For detailed information on how to use the shell as a command interpreter and a programming language, refer to Chapter 7. Complete information about shell programming is available in a separate document, *Shell Commands and Programming*.

Customizing Your Computing Environment

This section deals with another control provided by the shell: your environment. When you log in to the UNIX system, the shell automatically sets up a computing environment for you. The default environment set up by the shell includes these variables:

HOME	your login directory
LOGNAME	your login name
PATH	route the shell takes to search for executable files or commands (typically **PATH:/bin:/usr/bin**)

The **PATH** variable tells the shell where to look for the executable program invoked by a command. Therefore it is used every time you issue a command. If you have executable programs in more than one directory, you will want all of them to be searched by the shell to make sure every command can be found.

You can use the default environment supplied by your system or you can tailor an environment to meet your needs. If you choose to modify any part of your environment, you can use either of two methods to do so. If you want to change a part of your environment only for the duration of your current computing session, specify your changes in a command line (see Chapter 7 for details). However, if you want to use a different environment (not the default environment) regularly, you can specify your changes in a file that will set up the desired environment for you automatically every time you log in. This file must be called **.profile** and must be located in your home directory.

The **.profile** typically performs some or all of the following tasks: checks for mail; sets data parameters, terminal settings, and tab stops; assigns a character or character string as your login prompt; and assigns the erase and kill functions to keys. You can define as few or as many tasks as you want in your **.profile**. You can also change parts of it at any time. For instructions on modifying a **.profile**, see "Modifying Your Login Environment" in Chapter 7.

Now check to see whether or not you have a **.profile**. If you are not already in your home directory, **cd** to it. Then examine your **.profile** by issuing this command:

cat .profile

If you have a **.profile**, its contents will appear on your screen. If you do not have a **.profile** you can create one with a text editor, such as **ed** or **vi**. (See "Modifying Your Login Environment" in Chapter 7 for instructions.)

Programming in the Shell

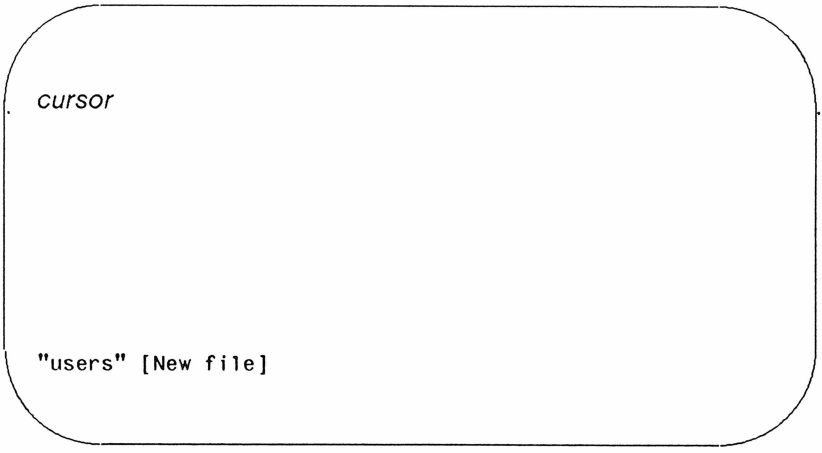
The shell is not only the command language interpreter; it is also a command level programming language. This means that instead of always using the shell strictly as a liaison between you and the computer, you can also program it to repeat sequences of instructions automatically. To do this, you must create executable files containing lists of commands. These files are called shell procedures or shell scripts. Once you have a shell script for a particular task, you can simply request that the shell read and execute the contents of the script whenever you want to perform that task.

Like other programming languages, the shell provides such features as variables, control structures, subroutines, and parameter passing. These features enable you to create your own tools by linking together system commands.

For example, you can combine three UNIX system programs (the **date**, **who**, and **wc** commands) into a simple shell script called **users** that tells you the current date and time, and how many users are working on your system. If you use the **vi** editor (described in Chapter 6) to create your script, you can follow this procedure. First, create the file **users** with the editor by typing

vi users <CR>

The editor will draw a blank page on your screen and wait for you to enter text.



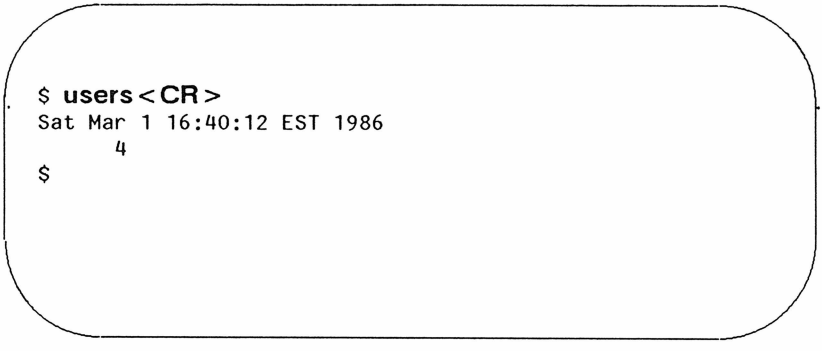
Enter the three UNIX system commands on one line:

date; who ¦ wc -l

Then write and quit the file. Make **users** executable by adding execute permission with the **chmod** command.

chmod ug+x users<CR>

Now try running your new command. The following screen shows the kind of output you will get.



The output tells you that four users were logged in on the system when you typed the command at 16:40 on Saturday, March 1, 1986.

For step-by-step instructions on writing shell scripts and information about more sophisticated shell programming techniques, see Chapter 7, "Shell Tutorial."

Communicating Electronically

As a UNIX system user, you can send messages or transmit information stored in files to other users who work on your system or another UNIX system. To do so, you must be logged in on a UNIX system that is capable of communicating with the UNIX system to which you want to send information. The command you use to send information depends on what you are sending. This guide introduces you to these communication programs:

mail This command allows you to send messages or files to other UNIX system users, using their login names as addresses. It also allows you to receive messages sent by other users. **mail** holds messages and lets the recipient read them at his or her convenience.

mailx This command is a sophisticated, more powerful version of **mail**. It offers a number of options for managing the electronic mail you send and receive.

uucp This command is used to send files from one UNIX system to another. (Its name is an acronym for UNIX to UNIX system copy.) You can use **uucp** to send a file to a directory you specify on a remote computer. When the file has been transferred, the owner of the directory is notified of its arrival by **mail**.

uuto/uupick These commands are used to send and retrieve files. You can use the **uuto** command to send a file(s) to a public directory; when it is available, the recipient is notified by mail that the file(s) has arrived. The recipient then can use the **uupick** command to copy the file(s) from the public directory to a directory of choice.

uux This command lets you execute commands on a remote computer. It gathers files from various computers, executes the specified command on these files, and sends the standard output to a file on the specified computer.

Programming in the System

The UNIX system provides a powerful and convenient environment for programming and software development, using the C programming language, FORTRAN-77, BASIC, Pascal, and COBOL. As well, the UNIX system provides some sophisticated tools designed to make software development easier and to provide a systematic approach to programming.

For information on available UNIX system programming languages, see the *Product Overview* or *Documentation Roadmap*.

For information on the general topic of programming in the UNIX system environment, see the *Programmer's Guide*. Besides supplementing texts on programming languages, the *Programmer's Guide* provides tutorials on the following five tools:

SCCS	Source Code Control System
RJE	Remote Job Entry (not available on all UNIX systems)
make	maintains programs
lex	generates programs for simple lexical tasks
yacc	generates parser programs

[This page left blank.]

5. LINE EDITOR TUTORIAL

Chapter 5: Line Editor Tutorial (ed)

Introducing the Line Editor 5-1

Suggestions for Using this Tutorial 5-3

Getting Started 5-4
- How to Enter **ed** 5-4
- How to Create Text 5-5
- How to Display Text 5-6
- How to Delete a Line of Text 5-8
- How to Move Up or Down in the File 5-10
- How to Save the Buffer Contents in a File 5-11
- How to Quit the Editor 5-12

Exercise 1 5-15

General Format of **ed** Commands 5-16

Line Addressing 5-17
- Numerical Addresses 5-17
- Symbolic Addresses 5-18
  - Symbolic Address of the Current Line 5-18
  - Symbolic Address of the Last Line 5-20
  - Symbolic Address of the Set of All Lines 5-20
  - Symbolic Address of Current Line to Last Line 5-21
  - Relative Addresses: Adding or Subtracting Lines from the Current Line 5-21
  - Character String Addresses 5-23
  - Specifying a Range of Lines 5-26
  - Specifying a Global Search 5-28

Exercise 2 5-32

Displaying Text in a File 5-33
Displaying Text Alone: the **p** Command 5-33
Displaying Text with Line Addresses: the **n** Command 5-34

Creating Text 5-36
Appending Text: the **a** Command 5-36
Inserting Text: the **i** Command 5-39
Changing Text: the **c** Command 5-41

Exercise 3 5-44

Deleting Text 5-46
Deleting Lines: the **d** Command 5-46
Undoing the Previous Command: the **u** Command 5-48
How to Delete in Text Input Mode 5-50
Escaping the Delete Function 5-50

Substituting Text 5-52
Substituting on the Current Line 5-53
Substituting on One Line 5-55
Substituting on a Range of Lines 5-55
Global Substitution 5-57

Exercise 4 5-61

Special Characters 5-63

Exercise 5 5-74

Moving Text 5-76
- Move Lines of Text 5-76
- Copy Lines of Text 5-79
- Joining Contiguous Lines 5-81
- Write Lines of Text to a File 5-82
- Problems 5-83
- Read in the Contents of a File 5-84

Exercise 6 5-86

Other Useful Commands and Information 5-87
- Help Commands 5-87
- Display Nonprinting Characters 5-90
- The Current File Name 5-91
- Escape to the Shell 5-93
- Recovering From System Interrupts 5-94
- Conclusion 5-95

Exercise 7 5-97

Answers to Exercises 5-97
- Exercise 1 5-98
- Exercise 2 5-100
- Exercise 3 5-103
- Exercise 4 5-106
- Exercise 5 5-109
- Exercise 6 5-112
- Exercise 7 5-115

Introducing the Line Editor

This chapter is a tutorial on the line editor, **ed**. **ed** is versatile and requires little computer time to perform editing tasks. It can be used on any type of terminal. The examples of command lines and system responses in this chapter will apply to your terminal, whether it is a video display terminal or a paper printing terminal. The **ed** commands can be typed in at your terminal or they can be used in a shell program (see Chapter 7, "Shell Tutorial").

ed is a line editor; during editing sessions it is always pointing at a single line in the file called the current line. When you access an existing file, **ed** makes the last line the current line so you can start appending text easily. Unless you specify the number of a different line or range of lines, **ed** will perform each command you issue on the current line. In addition to letting you change, delete, or add text on one or more lines, **ed** allows you to add text from another file to the buffer.

During an editing session with **ed**, you are altering the contents of a file in a temporary buffer, where you work until you have finished creating or correcting your text. When you edit an existing file, a copy of that file is placed in the buffer and your changes are made to this copy. The changes have no effect on the original file until you instruct **ed**, by using the write command, to move the contents of the buffer into the file.

After you have read through this tutorial and tried the examples and exercises, you will have a good working knowledge of **ed**. The following basics are included:

- entering the line editor **ed**, creating text, writing the text to file, and quitting **ed**
- addressing particular lines of the file and displaying lines of text
- deleting text
- substituting new text for old text
- using special characters as shortcuts in search and substitute patterns

- moving text around in the file, as well as other useful commands and information

Suggestions for Using this Tutorial

The commands discussed in each section are reviewed at the end of that section. A summary of all **ed** commands introduced in this chapter is found in Appendix C, where they are listed by topic.

At the end of some sections, exercises are given so you can experiment with the commands. The answers to all exercises are at the end of this chapter.

The notation conventions used in this chapter are those used throughout this *Guide*. They are described in the Preface.

Getting Started

The best way to learn **ed** is to log in to the UNIX system and try the examples as you read this tutorial. Do the exercises; do not be afraid to experiment. As you experiment and try out **ed** commands, you will learn a fast and versatile method of text editing.

In this section you will learn the commands used to:

- enter **ed**
- append text
- move up or down in the file to display a line of text
- delete a line of text
- write the buffer to a file
- quit **ed**

How to Enter ed

To enter the line editor, type **ed** and a file name:

ed *filename* **<CR>**

Choose a name that reflects the contents of the file. If you are creating a new file, the system responds with a question mark and the file name:

```
$ ed new-file <CR>
?new-file
```

If you going to edit an existing file, **ed** responds with the number of characters in the file:

```
$ ed old-file <CR>
235
```

How to Create Text

The editor receives two types of input, editing commands and text, from your terminal. To avoid confusing them, **ed** recognizes two modes of editing work: command mode and text input mode. When you work in command mode, any characters you type are interpreted as commands. In input mode, any characters you type are interpreted as text to be added to a file.

Whenever you enter **ed** you are put into command mode. To create text in your file, change to input mode by typing **a** (for append), on a line by itself, and pressing the RETURN key:

a<CR>

Now you are in input mode; any characters you type from this point wil be added to your file as text. Be sure to type **a** on a line by itself; if you do not, the editor will not execute your command.

After you have finished entering text, type a period on a line by itself. This takes you out of the text input mode and returns you to the command mode. Now you can give **ed** other commands.

The following example shows how to enter **ed**, create text in a new file called **try-me**, and quit text input mode with a period.

```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is the second line,<CR>
and this is the third line.<CR>
.<CR>
```

Notice that **ed** does not give a response to the period; it just waits for a new command. If **ed** does not respond to a command, you may have forgotten to type a period after entering text and may still be in text input mode. Type a period and press the RETURN key at the beginning of a line to return to command mode. Now you can execute editing commands. For example, if you have added some unwanted characters or lines to your text, you can delete them once you have returned to command mode.

How to Display Text

To display a line of a file, type **p** (for print) on a line by itself. The **p** command prints the current line, that is, the last line on which you worked. Continue with the previous example. You have just typed a period to exit input mode. Now type the **p** command to see the current line.

```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is the second line,<CR>
and this is the third line.<CR>
.<CR>
p<CR>
and this is the third line.
```

You can print any line of text by specifying its line number (also known as the address of the line). The address of the first line is 1; of the second, 2; and so on. For example, to print the second line in the file **try-me**, type:

```
2p<CR>
This is the second line,
```

You can also use line addresses to print a span of lines by specifying the addresses of the first and last lines of the section you want to see, separated by a comma. For example, to print the first three lines of a file, type:

1,3p<CR>

You can even print the whole file this way. For example, you can display a twenty-line file by typing **1,20p**. If you do not know the address of the last line in your file, you can substitute a **$** sign, **ed** symbol for the address of the last line. (These conventions are discussed in detail in the section "Line Addressing.")

```
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
```

If you forget to quit text input mode with a period, you will add text that you do not want. Try to make this mistake. Add another line of text to your **try-me** file and then try the **p** command without quitting text input mode. Then quit text input mode and print the entire file.

```
p<CR>
and this is the third line.
a<CR>
This is the fourth line.<CR>
p<CR>
.<CR>
1,$p<CR>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
p
```

What did you get? The next section will explain how to delete the unwanted line.

How to Delete a Line of Text

To delete text, you must be in the command mode of **ed**. Typing **d** deletes the current line. Try this command on the last example to remove the unwanted line containing `p`. Display the current line (**p** command), delete it (**d** command), and display the remaining lines in the file (**p** command). Your screen should look like this:

```
p<CR>
p
d<CR>
1,$p<CR>
This is the first line of text.
This is a second line,
and this is the third line.
This is the fourth line.
```

ed does not send you any messages to confirm that you have deleted text. The only way you can verify that the **d** command has succeeded is by printing the contents of your file with the **p** command. To receive verification of your deletion, you can put the **d** and **p** together on one command line. If you repeat the previous example with this command, your screen should look like this:

```
p<CR>
p
dp<CR>
This is the fourth line.
```

How to Move Up or Down in the File

To display the line below the current line, press the RETURN key while in command mode. If there is no line below the current line, **ed** responds with a ? and continues to treat the last line of the file as the current line. To display the line above the current line, press the minus key (−).

The following screen provides examples of how both of these commands are used:

```
p<CR>
This is the fourth line.
-<CR>
and this is the third line.
-<CR>
This is a second line,
-<CR>
This is the first line of text.
<CR>
This is a second line,
<CR>
and this is the third line.
```

Notice that by typing −<CR> or <CR>, you can display a line of text without typing the **p** command. These commands are also line addresses. Whenever you type a line address and do not follow it with a command, **ed** assumes that you want to see the line you have specified. Experiment with these commands: create some text, delete a line, and display your file.

How to Save the Buffer Contents in a File

As we discussed earlier, during an editing session, the system holds your text in a temporary storage area called a buffer. When you have finished editing, you can save your work by writing it from the temporary buffer to a permanent file in the computer's memory. By writing to a file, you are simply putting a copy of the contents of the buffer into the file. The text in the buffer is not disturbed, and you can make further changes to it.

NOTE: It is a good idea to write the buffer text into your file frequently. If an interrupt occurs (such as an accidental loss of power to your terminal), you may lose the material in the buffer, but you will not lose the copy written to your file.

To write your text to a file, enter the **w** command. You do not need to specify a file name; simply type **w** and press the RETURN key. If you have just created new text, **ed** creates a file for it with the name you specified when you entered the editor. If you have edited an existing file, the **w** command writes the contents of the buffer to that file by default.

If you prefer, you can specify a new name for your file as an argument on the **w** command line. Be careful not to use the name of a file that already exists unless you want to replace its contents with the contents of the current buffer. **ed** will not warn you about an existing file; it will simply overwrite that file with your buffer contents.

For example, if you decide you would prefer the **try-me** file to be called **stuff**, you can rename it:

```
$ ed try-me<CR>
? try-me
a<CR>
This is the first line of text.<CR>
This is the second line,<CR>
and this is the third line.<CR>
.
w stuff <CR>
85
```

Notice the last line of the screen. This is the number of characters in your text. When the editor reports the number of characters in this way, the write command has succeeded.

How to Quit the Editor

When you have completed editing your text, write it from the buffer into a file with the w command. Then leave the editor and return to the shell by typing **q** (for quit).

```
w<CR>
85
q<CR>
$
```

The system responds with a shell prompt. At this point the editing buffer vanishes. If you have not executed the write command,

your text in the buffer has also vanished. If you did not make any changes to the text during your editing session, no harm is done. However, if you did make changes, you could lose your work in this way. Therefore, if you type **q** after changing the file without writing it, **ed** warns you with a ?. You then have a chance to write and quit.

```
q<CR>
?
w<CR>
85
q<CR>
$
```

If, instead of writing, you insist on typing **q** a second time, **ed** assumes you do not want to write the buffer's contents to your file and returns you to the shell. Your file is left unchanged and the contents of the buffer are wiped out.

You now know the basic commands needed to create and edit a file using **ed**. Figure 5-1 summarizes these commands.

Command	Function
ed *file*	**enter ed** to edit *file*
a	append text after the current line
.	quit text input mode and return to **ed** command mode.
p	print text on your terminal
d	delete text
<CR>	display the next line in the buffer (literally, carriage return)
+	display the next line in the buffer
–	display the previous line in the buffer
w	write the contents of the buffer to the file
q	quit **ed** and return to the shell

Figure 5-1: Summary of **ed** Editor Commands

Exercise 1

Answers for all the exercises in this chapter are found at the end of the chapter. However, they are not necessarily the only possible correct answers. Any method that enables you to perform a task specified in an exercise is correct, even if it does not match the answer given.

1-1. Enter **ed** with a file named **junk**. Create a line of text containing **Hello World**, write it to the file and quit **ed**.

Now use **ed** to create a file called **stuff**. Create a line of text containing two words, **Goodbye world**, write this text to the file, and quit **ed**.

1-2. Enter **ed** again with the file named **junk**. What was the editor's response? Was the character count for it the same as the character count reported by the **w** command in Exercise 1-1?

Display the contents of the file. Is that your file **junk**?

How can you return to the shell? Try **q** without writing the file. Why do you think the editor allowed you to quit without writing to the buffer?

1-3. Enter **ed** with the file **junk**. Add a line:

Wendy's horse came through the window.

Since you did not specify a line address, where do you think the line was added to the buffer? Display the contents of the buffer. Try quitting the buffer without writing to the file. Try writing the buffer to a different file called **stuff**. Notice that **ed** does not warn you that a file called **stuff** already exists. You have erased the contents of **stuff** and replaced them with new text.

General Format of ed Commands

ed commands have a simple and regular format:

[*address1*[,*address2*]]*command*[*argument*] <**CR**>

The brackets around *address1*, *address2*, and *argument* show that these are optional. The brackets are not part of the command line.

address1,*address2*
: The addresses give the position of lines in the buffer. *Address1* through *address2* gives you a range of lines that will be affected by the *command*. If *address2* is omitted, the command will affect only the line specified by *address1*.

command
: The *command* is one character and tells the editor what task to perform.

argument
: The *arguments* to a *command* are those parts of the text that will be modified, or a file name, or another line address.

This format will become clearer to you when you begin to experiment with the **ed** commands.

Line Addressing

A line address is a character or group of characters that identifies a line of text. Before **ed** can execute commands that add, delete, move, or change text, it must know the line address of the affected text. Type the line address before the command:

[*address1*],[*address2*]*command* <**CR**>

Both *address1* and *address2* are optional. Specify *address1* alone to request action on a single line of text; both *address1* and *address2* to request a span of lines. If you do not specify any *address*, **ed** assumes that the line address is the current line.

The most common ways to specify a line address in **ed** are:

- by entering line numbers (assuming that the lines of the files are consecutively numbered from 1 to *n*, beginning with the first line of the file)
- by entering special symbols for the current line, last line, or a span of lines
- by adding or subtracting lines from the current line
- by searching for a character string or word on the desired line

You can access one line or a span of lines, or make a global search for all lines containing a specified character string. (A character string is a set of successive characters, such as a word.)

Numerical Addresses

ed gives a numerical address to each line in the buffer. The first line of the buffer is 1, the second line is 2, and so on, for each line in the buffer. Any line can be accessed by **ed** with its line address number. To see how line numbers address a line, enter **ed** with the file **try-me** and type a number.

```
$ ed try-me<CR>
110
1<CR>
This is the first line of text.
3<CR>
and this is the third line.
```

Remember that **p** is the default command for a line address specified without a command. Because you gave a line address, **ed** assumes you want that line displayed on your terminal.

Numerical line addresses frequently change in the course of an editing session. Later in this chapter you will create lines, delete lines, or move a line to a different position. This will change the line address numbers of some lines. The number of a specific line is always the current position of that line in the editing buffer. For example, if you add five lines of text between line 5 and 6, line 6 becomes line 11. If you delete line 5, line 6 becomes line 5.

Symbolic Addresses

Symbolic Address of the Current Line

The current line is the line most recently acted on by any **ed** command. If you have just entered **ed** with an existing file, the current line is the last line of the buffer. The symbol for the address of the current line is a period. Therefore you can display the current line simply by typing a period (.) and pressing the RETURN key.

Try this command in the file **try-me**:

```
$ ed try-me<CR>
110
.<CR>
This is the fourth line.
```

The . is the address. Because a command is not specified after the period, **ed** executes the default command **p** and displays the line found at this address.

To get the line number of the current line, type the following command:

.= <CR>

ed responds with the line number. For example, in the **try-me** file, the current line is 4.

```
.<CR>
This is the fourth line.
.= <CR>
4
```

Symbolic Address of the Last Line

The symbolic address for the last line of a file is the **$** sign. To verify that the **$** sign accesses the last line, access the **try-me** file with **ed** and specify this address on a line by itself. (Keep in mind that when you first access a file, your current line is always the last line of the file.)

```
$ ed try-me<CR>
110
.<CR>
This is the fourth line.
$<CR>
This is the fourth line.
```

Remember that the **$** address within **ed** is not the same as the $ prompt from the shell.

Symbolic Address of the Set of All Lines

When used as an address, a comma (,) refers to all the lines of a file, from the first through the last line. It is an abbreviated form of the string mentioned earlier that represents all lines in a file, **1,$**. Try this shortcut to print the contents of **try-me**:

```
,p<CR>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line.
```

Symbolic Address of the Current Line through the Last Line

The semi-colon (;) represents a set of lines beginning with the current line and ending with the last line of a file. It is equivalent to the symbolic address .,$. Try it with the file **try-me**:

```
2p<CR>
This is the second line,
;p<CR>
This is the second line,
and this is the third line.
This is the fourth line.
```

Relative Addresses: Adding or Subtracting Lines from the Current Line

You may often want to address lines with respect to the current line. You can do this by adding or subtracting a number of lines from the current line with a plus () or a minus (−) sign. Addresses derived in this way are called relative addresses. To experiment with relative line addresses, add several more lines to your file **try-me**, as shown in the following screen. Also, write the

buffer contents to the file so your additions will be saved:

```
$ ed try-me<CR>
110
.<CR>
This is the fourth line.
a<CR>
five<CR>
six<CR>
seven<CR>
eight<CR>
nine<CR>
ten<CR>
.<CR>
w<CR>
140
```

Now try adding and subtracting line numbers from the current line.

```
4<CR>
This is the fourth line.
3<CR>
seven
-5<CR>
This is a second line,
```

What happens if you ask for a line address that is greater than the last line, or if you try to subtract a number greater than the current line number?

```
5<CR>
five
-6<CR>
?
.=<CR>
5
7<CR>
?
```

Notice that the current line remains at line 5 of the buffer. The current line changes only if you give **ed** a correct address. The **?** response means there is an error. "Other Useful Commands and Information," at the end of this chapter, explains how to get a help message that describes the error.

Character String Addresses

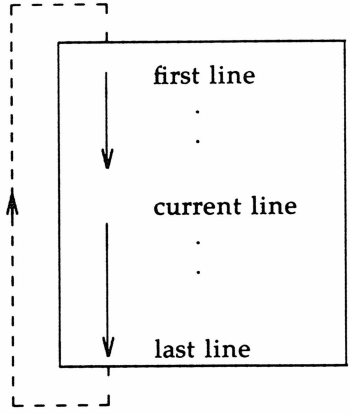
You can search forward or backward in the file for a line containing a particular character string. To do so, specify a string, preceded by a delimiter.

Delimiters mark the boundaries of character strings; they tell **ed** where a string starts and ends. The most common delimiter is / (slash), used in the following format:

/pattern

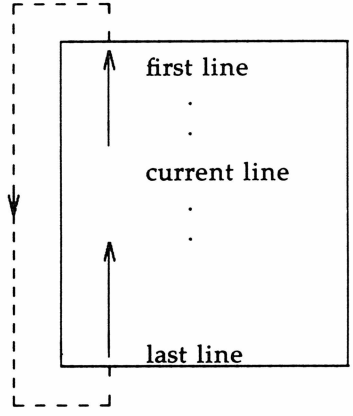
When you specify a pattern preceded by a / (slash), **ed** begins at the current line and searches forward (down through subsequent lines in the buffer) for the next line containing the *pattern*. When the search reaches the last line of the buffer, **ed** wraps around to the beginning of the file and continues its search from line 1.

The following rectangle represents the editing buffer. The path of the arrows shows the search initiated by a / :



Another useful delimiter is **?**. If you specify a pattern preceded by a **?**, (**?***pattern*), **ed** begins at the current line and searches backward (up through previous lines in the buffer) for the next line containing the *pattern*. If the search reaches the first line of the file, it will wrap around and continue searching upward from the last line of the file.

The following rectangle represents the editing buffer. The path of the arrows shows the search initiated by a **?** :



Experiment with these two methods of requesting address searches on the file **try-me**. What happens if **ed** does not find the specified character string?

```
$ ed try-me <CR>
140
.<CR>
ten
?first<CR>
This is the first line of text.
/fourth<CR>
This is the fourth line.
/junk<CR>
?
```

In this example, **ed** found the specified strings **first** and **fourth**. Then, because no command was given with the address, it executed the **p** command by default, displaying the lines it had found. When **ed** cannot find a specified string (such as **junk**), it responds with a ? .

You can also use the / (slash) to search for multiple occurrences of a pattern without typing it more than once. First, specify the pattern by typing */pattern*, as usual. After **ed** has printed the first occurrence, it waits for another command. Type / and press the RETURN key; **ed** will continue to search forward through the file for the last *pattern* specified. Try this command by searching for the word **line** in the file **try-me**:

```
.<CR>
This is the first line of text.
/line<CR>
This is the second line,
/<CR>
and this is the third line.
/<CR>
This is the fourth line.
/<CR>
This is the first line of text.
```

Notice that after **ed** has found all occurrences of the *pattern* between the line where you requested a search and the end of the file, it wraps around to the beginning of the file and continues searching.

Specifying a Range of Lines

There are two ways to request a group of lines. You can specify a range of lines, such as *address1* through *address2*, or you can specify a global search for all lines containing a specified pattern.

The simplest way to specify a range of lines is to use the line numbers of the first and last lines of the range, separated by a comma. Place this address before the command. For example, if you want to display lines 2 through 7 of the editing buffer, give *address1* as 2 and *address2* as 7 in the following format:

2,7p<CR>

Try this on the file **try-me**:

```
2,7p<CR>
This is the second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

Did you try typing **2,7** without the **p**? What happened? If you do not add the **p** command, **ed** prints only *address2*, the last line of the range of addresses.

Relative line addresses can also be used to request a range of lines. Be sure that *address1* precedes *address2* in the buffer. Relative addresses are calculated from the current line, as the following example shows:

```
4<CR>
This is the fourth line
-2,3p<CR>
This is the second line,
and this is the third line.
This is the fourth line.
five
six
seven
```

Specifying a Global Search

There are two commands that do not follow the general format of **ed** commands: **g** and **v**. These are global search commands that specify addresses with a character string (*pattern*). The **g** command searches for all lines containing the string *pattern* and performs the *command* on those lines. The **v** command searches for all lines that do not contain the *pattern* and performs the *command* on those lines.

The general format for these commands is:

g/*pattern*/*command*<**CR**>
v/*pattern*/*command*<**CR**>

Try these commands by using them to search for the word **line** in **try-me**:

```
g/line/p<CR>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line
```

```
v/line/p<CR>
five
six
seven
eight
nine
ten
```

Notice the function of the **v** command: it finds all the lines that do not contain the word specified in the command line (**line**).

Once again, the default command for the lines addressed by **g** or **v** is **p**; you do not need to include a **p** as the last delimiter on your command line.

```
g/line<CR>
This is the first line of text.
This is the second line,
and this is the third line.
This is the fourth line
```

However, if you are giving line addresses to be used by other **ed** commands, you need to include beginning and ending delimiters. You can use any of the methods discussed in this section to specify line addresses for **ed** commands. Figure 5-2 summarizes the symbols and commands available for addressing lines.

Address	Description
n...	the number of a line in the buffer
.	the current line (the line most recently acted on by an **ed** command)
.=	the command used to request the line number of the current line
$	the last line of the file
,	the set of lines from line 1 through the last line
;	the set of lines from the current line through the last line
n	the line that is located *n* lines after the current line
– *n*	the line that is located *n* lines before the current line
/*abc*	the command used to search forward in the buffer for the first line that contains the pattern *abc*
?*abc*	the command used to search backward in the buffer for the first line that contains the pattern *abc*
g/*abc*	the set of all lines that contain the pattern *abc*
v/*abc*	the set of all lines that do NOT contain the pattern *abc*

Figure 5-2: Summary of Line Addressing

Exercise 2

2-1. Create a file called **towns** with the following lines:

My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
I lost my heart in
San Francisco
I lost $$ in
Las Vegas

2-2. Display line 3.

2-3. If you specify a range of lines with the relative address −2,3p, what lines are displayed ?

2-4. What is the current line number? Display the current line.

2-5. What does the last line say?

2-6. What line is displayed by the following request for a search?

?town < CR >

After **ed** responds, type this command alone on a line:

? < CR >

What happened?

2-7. Search for all lines that contain the pattern in. Then search for all lines that do NOT contain the pattern in.

Displaying Text in a File

ed provides two commands for displaying lines of text in the editing buffer: **p** and **n**.

Displaying Text Alone: the p Command

You have already used the **p** command in several examples. You are probably now familiar with its general format:

[*address1*,*address2*]**p** < **CR** >

p does not take arguments. However, it can be combined with a substitution command line. This will be discussed later in this chapter.

Experiment with the line addresses shown in Figure 5-3 on a file in your home directory. Try the **p** command with each address and see if **ed** responds as described in the figure.

Specify this Address	Check for this Response
1,$p < CR >	**ed** should display the entire file on your terminal.
– 5p < CR >	**ed** should move backward five lines from the current line and display the line found there.
2p < CR >	**ed** should move forward two lines from the current line and display the line found there.
1,/*x*/p < CR >	**ed** displays the set of lines from line one through the first line after the current line that contains the character *x*. It is important to enclose the letter *x* between slashes so that **ed** can distinguish between the search pattern address (*x*) and the **ed** command (**p**).

Figure 5-3: Sample Addresses for Displaying Text

Displaying Text with Line Addresses: the n Command

The **n** command displays text and precedes each line with its numerical line address. It is helpful when you are deleting, creating, or changing lines. The general command line format for **n** is the same as that for **p**.

[*address1,address2*]**n** < **CR** >

Like **p**, **n** does not take arguments, but it can be combined with the substitute command.

Try running **n** on the **try-me** file:

```
$ ed try-me<CR>
140
1,$n<CR>
1       This is the first line of text.
2       This is the second line,
3       and this is the third line.
4       This is the fourth line.
5       five
6       six
7       seven
8       eight
9       nine
10      ten
```

Figure 5-4 summarizes the **ed** commands for displaying text.

Command	Function
p	displays specified lines of text in the editing buffer on your terminal
n	displays specified lines of text in the editing buffer with their numerical line addresses on your terminal

Figure 5-4: Summary of Commands for Displaying Text

Creating Text

ed has three basic commands for creating new lines of text:

- **a** append text
- **i** insert text
- **c** change text

Appending Text: the a Command

The append command, **a**, allows you to add text AFTER the current line or a specified address in the file. You have already used this command in the "Getting Started" section of this chapter. The general format for the append command line is:

[*address1*]**a** < **CR** >

Specifying an address is optional. The default value of *address1* is the current line.

In previous exercises, you used this command with the default address. Now try using different line numbers for *address1*. In the following example, a new file called **new-file** is created. In the first append command line, the default address is the current line. In the second append command line, line 1 is specified as *address1*. The lines are displayed with **n** so that you can see their numerical line addresses. Remember, the append mode is ended by typing a period (.) on a line by itself.

```
$ ed new-file<CR>
?new-file
a<CR>
Create some lines
of text in
this file.
.<CR>
1,$n<CR>
1       Create some lines
2       of text in
3       this file.
1a<CR>
This will be line 2<CR>
This will be line 3<CR>
.<CR>
1,$n<CR>
1       Create some lines
2       This will be line 2
3       This will be line 3
4       of text in
5       this file.
```

Notice that after you append the two new lines, the line that was originally line 2 (`of text in`) becomes line 4.

You can take shortcuts to places in the file where you want to append text by combining the append command with symbolic addresses. The following three command lines allow you to move through and add to the text quickly in this way.

.a<CR> appends text after the current line

$a<CR> appends text after the last line of the file

0a<CR> appends text before the first line of the file (at a symbolic address called line 0)

To try using these addresses, create a one-line file called **lines** and type the examples shown in the following screens. (The examples appear in separate screens for easy reference only; it is not necessary to access the **lines** file three times to try each append symbol. You can access **lines** once and try all three consecutively.)

```
$ ed lines<CR>
?lines
a<CR>
This is the current line.<CR>
.<CR>
p<CR>
This is the current line.
.a<CR>
This line is after the current line.<CR>
.<CR>
-1,.p<CR>
This is the current line.
This line is after the current line.
```

```
$a<CR>
This is the last line now.<CR>
.<CR>
$<CR>
This is the last line now.
```

```
0a<CR>
This is the first line now.<CR>
This is the second line now.<CR>
The line numbers change<CR>
as lines are added.<CR>
.<CR>
1,4n<CR>
1               This is the first line now.
2               This is the second line now.
3               The line numbers change
4               as lines are added.
```

Because the append command creates text after a specified address, the last example refers to the line before line 1 as the line after line 0. To avoid such circuitous references, use another command provided by the editor: the insert command, **i**.

Inserting Text: the i Command

The insert command (**i**), allows you to add text BEFORE a specified line in the editing buffer. The general command line format for **i** is the same as that for **a**.

[*address1*]**i**<CR>

As with the append command, you can insert one or more lines of text. To quit input mode, you must type a period (.) alone on a line.

Create a file called **insert** in which you can try the insert command (**i**):

```
$ ed insert<CR>
?insert
a<CR>
Line 1<CR>
Line 2<CR>
Line 3<CR>
Line 4<CR>
.<CR>
w<CR>
69
```

Now insert one line of text above line 2 and another above line 1. Use the **n** command to display all the lines in the buffer:

```
2i<CR>
This is the new line 2.<CR>
.<CR>
1,$n<CR>
1       Line 1
2       This is the new line 2.
3       Line 2
4       Line 3
5       Line 4
1i<CR>
This is the beginning.<CR>
.<CR>
1,$n<CR>
1       In the beginning
2       Line 1
3       Now this is line 2
4       Line 2
5       Line 3
6       Line 4
```

Experiment with the insert command by combining it with symbolic line addresses, as follows:

- **.i<CR>**
- **$i<CR>**

Changing Text: the c Command

The change text command (**c**) erases all specified lines and allows you to create one or more lines of text in their place. Because **c** can erase a range of lines, the general format for the command line includes two addresses.

[*address1*,*address2*]**c<CR>**

The change command puts you in text input mode. To leave input mode, type a period alone on a line.

Address1 is the first and *address2* is the last of the range of lines to be replaced by new text. To erase one line of text, specify only *address1*. If no address is specified, **ed** assumes the current line is the line to be changed.

Now create a file called **change** in which you can try this command. After entering the text shown in the screen, change lines one through four by typing **1,4c**:

```
1,5n<CR>
1       line 1
2       line 2
3       line 3
4       line 4
5       line 5
1,4c<CR>
Change line 1<CR>
and lines 2 through 4<CR>
.<CR>
1,$n<CR>
1       change line 1
2       and lines 2 through 4
3       line 5
```

Now experiment with **c** and try to change the current line:

```
.<CR>
line 5
c<CR>
This is the new line 5.
.<CR>
.<CR>
This is the new line 5.
```

If you are not sure whether you have left text input mode, it is a good idea to type another period. If the current line is displayed, you know you are in the command mode of **ed**.

Figure 5-5 summarizes the **ed** commands for creating text.

Command	Function
a	append text after the specified line in the buffer
i	insert text before the specified line in the buffer
c	change the text on the specified line(s) to new text
.	quit text input mode and return to **ed** command mode

Figure 5-5: Summary of Commands for Creating Text

Exercise 3

3-1. Create a new file called **ex3**. Instead of using the append command to create new text in the empty buffer, try the insert command. What happens?

3-2. Enter **ed** with the file **towns**. What is the current line?

Insert above the third line:

Illinois < CR >

Insert above the current line:

or < CR >
Naperville < CR >

Insert before the last line:

hotels in < CR >

Display the text in the buffer preceded by line numbers.

3-3. In the file **towns**, display lines 1 through 5 and replace lines 2 through 5 with:

London < CR >

Display lines 1 through 3.

3-4. After you have completed exercise 3-3, what is the current line?

Find the line of text containing:

```
Toledo
```

Replace

```
Toledo
```

with

Peoria

Display the current line.

3-5 With one command line search for and replace:

```
New York
```

with:

Iron City

Deleting Text

This section discusses two types of commands for deleting text in **ed**. One type is to be used when you are working in command mode: **d** deletes a line and **u** undoes the last command. The other type of command is to be used in text input mode: <#> (the pound sign) deletes a character and <@> (the at sign) kills a line. The delete keys that are used in input mode are the same keys you use to delete text that you enter after a shell prompt. They are described in detail in "Correcting Typing Errors" in Chapter 2.

Deleting Lines: the d Command

You have already deleted lines of text with the delete command (**d**) in the "Getting Started" section of this chapter.

The general format for the **d** command line is:

[*address1*,*address2*]**d** <**CR**>

You can delete a range of lines (*address1* through *address2*) or you can delete one line only (*address1*). If no address is specified, **ed** deletes the current line.

The next example displays lines one through five and then deletes lines two through four:

```
1,5n<CR>
1           1 horse
2           2 chickens
3           3 ham tacos
4           4 cans of mustard
5           5 bails of hay
2,4d<CR>
1,$n<CR>
1           1 horse
2           5 bails of hay
```

How can you delete only the last line of a file? Using a symbolic line address makes this easy:

$d<CR>

How can you delete the current line? One of the most common errors in **ed** is forgetting to type a period to leave text input mode. When this happens, unwanted text may be added to the buffer. In the next example, a line containing a print command (**1,$p**) is accidentally added to the text before the user leaves input mode. Because this line was the last one added to the text, it becomes the current line. The symbolic address . is used to delete it.

```
a<CR>
Last line of text<CR>
1,$p<CR>
.<CR>
p<CR>
1,$p
.d<CR>
p<CR>
Last line of text.
```

Before experimenting with the delete command, you may first want to learn about the undo command, **u**.

Undoing the Previous Command: the u Command

The command **u** (short for undo) nullifies the last command and restores any text changed or deleted by that command. It takes no addresses or arguments. The format is:

u<CR>

One purpose for which the **u** command is useful is to restore text you have mistakenly deleted. If you delete all the lines in a file and then type **p, ed** will respond with a ? since there are no more lines in the file. Use the **u** command to restore them.

```
1,$p<CR>
This is the first line.
This is the middle line.
This is the last line.
1,$d<CR>
p<CR>
?
u<CR>
p<CR>
This is the last line.
```

Now experiment with **u**: use it to undo the append command.

```
.<CR>
This is the only line of text
a<CR>
Add this line<CR>
.<CR>
1,$p<CR>
This is the only line of text
Add this line
u<CR>
1,$p<CR>
This is the only line of text
```

NOTE: **u** cannot be used to undo the write command (**w**) or the quit command (**q**). However, **u** can undo an undo command (**u**).

How to Delete in Text Input Mode

While in text input mode, you can correct the current line of input with the same keys you use to correct a shell command line. By default, there are two keys available to correct text. The @ sign key kills the current line. The # sign key backs up over one character on the current line so you can retype it, thus effectively erasing the original character. (See "Correcting Typing Errors" in Chapter 2 for details.)

As mentioned in Chapter 2, you can reassign the line kill and character erase functions to other keys if you prefer. (See "Modifying Your Login Environment" in Chapter 7 for instructions.) If you have reassigned these functions, you must use the keys you chose while working in **ed**; the default keys (@ and #) will no longer work.

Escaping the Delete Function

You may want to include an @ sign or a # sign as a character of text. To avoid having these characters interpreted as delete commands, you must precede them with a \ (backslash), as shown in the following example.

```
a<CR>
leave San Francisco \@ 20:15 on flight \#347 <CR>
.<CR>
p<CR>
leave San Francisco @ 20:15 on flight #347
```

Figure 5-6 summarizes the **ed** commands and shell commands for deleting text in **ed**.

Command	Function
In command mode:	
<**d**>	delete one or more lines of text
<**u**>	undo the previous command
<@>	delete the current command line
In text input mode:	
<@>	delete the current line
<#> or <**BACKSPACE**>	delete the last character typed in

Figure 5-6: Summary of Commands for Deleting Text

Substituting Text

You can modify your text with a substitute command. This command replaces the first occurrence of a string of characters with new text. The general command line format is

[*address1*,*address2*]**s**/*old_text*/*new_text*/[*command*] < **CR** >

Each component of the command line is described below.

address1,*address2*
: The range of lines being addressed by **s**. The address can be one line, (*address1*), a range of lines (*address1* through *address2*), or a global search address. If no address is given, **ed** makes the substitution on the current line.

s
: The substitute command

/*old_text*
: The argument specifying the text to be replaced is usually delimited by slashes, but can be delimited by other characters such as a **?** or a period. It consists of the words or characters to be replaced. The command will replace the first occurrence of these characters that it finds in the text.

/*new_text*
: The argument specifying the text to replace *old_text*. It is delimited by slashes or the same delimiters used to specify the *old_text*. It consists of the words or characters that are to replace the *old_text*.

/*command*
: Any one of the following four commands:

g	Change all occurrences of *old_text* on the specified lines.
l	Display the last line of substituted text, including nonprinting characters. (See the last section of this chapter, "Other Useful Commands and Information.")
n	Display the last line of the substituted text preceded by its numerical line address.
p	Display the last line of substituted text.

Substituting on the Current Line

The simplest example of the substitute command is making a change to the current line. You do not need to give a line address for the current line.

s/*old_text*/*new_text*/**<CR>**

The next example contains a typing error. While the line that contains it is still the current line, you make a substitution to correct it. The old text is the **ai** of **airor** and the new text is **er**.

```
a<CR>
In the beginning, I made an airor.
.<CR>
.p<CR>
In the beginning, I made an airor.
s/ai/er<CR>
```

Notice that **ed** gives no response to the substitute command. To verify that the command has succeeded in this case, you either have to display the line with **p** or **n**, or include **p** or **n** as part of the substitute command line. In the following example, **n** is used to verify that the word **file** has been substituted for the word **toad**.

```
.p<CR>
This is a test toad
s/toad/file/n<CR>
1        This is a test file
```

However, **ed** allows you one shortcut: it prints the results of the command automatically, if you omit the last delimiter after the *new_text* argument:

```
.p<CR>
This is a test file
s/file/frog<CR>
This is a test frog
```

Substituting on One Line

To substitute text on a line that is not the current line, include an address in the command line, as follows:

[*address1*]**s**/*old_text*/*new_text*/<**CR**>

For example, in the following screen the command line includes an address for the line to be changed (line 1) because the current line is line 3:

```
1,3p<CR>
This is a pest toad
testing testing
come in toad
.<CR>
come in toad
1s/pest/test<CR>
This is a test toad
```

As you can see, **ed** printed the new line automatically after the change was made, because the last delimiter was omitted.

Substituting on a Range of Lines

You can make a substitution on a range of lines by specifying the first address (*address1*) through the last address (*address2*).

[*address1*,*address2*]**s**/*old_text*/*new_text*/<**CR**>

If **ed** does not find the pattern to be replaced on a line, no changes are made to that line.

In the following example, all the lines in the file are addressed for the substitute command. However, only the lines that contain the string **es** (the *old_text* argument) are changed.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
1,$s/es/ES/n<CR>
4        tESting 1, 2, 3
```

When you specify a range of lines and include **p** or **n** at the end of the substitute line, only the last line changed is printed.

To display all the lines in which text was changed, use the **n** or **p** command with the address **1,$**.

```
1,$n<CR>
1        This is a tESt toad
2        tESting testing
3        come in toad
4        tESting 1, 2, 3
```

Notice that only the first occurrence of **es** (on line 2) has been changed. To change every occurrence of a pattern, use the **g** command, described in the next section.

Global Substitution

One of the most versatile tools in **ed** is global substitution. By placing the **g** command after the last delimiter on the substitute command line, you can change every occurrence of a pattern on the specified lines. Try changing every occurrence of the string es in the last example. If you are following along, doing the examples as you read this, remember you can use **u** to undo the last substitute command.

```
u<CR>
1,$p<CR>
This is a test toad
testing, testing
come in toad
testing 1, 2, 3
1,$s/es/ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Another method is to use a global search pattern as an address instead of the range of lines specified by **1,$**.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/test/s/es/ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

If the global search pattern is unique and matches the argument *old_text* (text to be replaced), you can use an **ed** shortcut: specify the pattern once as the global search address and do not repeat it as an *old_text* argument. **ed** will remember the pattern from the search address and use it again as the pattern to be replaced.

g/*old_text*/**s**//*new_text*/**g**<**CR**>

NOTE: Whenever you use this shortcut, be sure to include two slashes (//) after the **s**.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
g/es/s//ES/g<CR>
1,$p<CR>
This is a tESt toad
tESting tESting
come in toad
tESting 1, 2, 3
```

Experiment with other search pattern addresses:

/pattern **<CR>**
?*pattern* **<CR>**
v/*pattern* **<CR>**

See what they do when combined with the substitute command. In the following example, the **v/***pattern* search format is used to locate lines that do not contain the pattern `testing`. Then the substitute command (**s**) is used to replace the existing pattern (`in`) with a new pattern (**out**) on those lines.

```
v/testing/s/in/out<CR>
This is a test toad
come out toad
```

Notice that the line `This is a test toad` was also printed, even though no substitution was made on it. When the last delimiter is omitted, all lines found with the search address are printed, regardless of whether or not substitutions have been made on them.

Now search for lines that do contain the pattern `testing` with the **g** command.

```
. g/testing/s//jumping<CR>
jumping testing
jumping 1, 2, 3
```

Notice that this command makes substitutions only for the first occurrence of the *pattern* (`testing`) in each line. Once again, the lines are displayed on your terminal because the last delimiter has been omitted.

Exercise 4

4-1. In your file **towns** change `town` to **city** on all lines but the line with `little town` on it.

The file should read:

My kind of city is
London
Like being no where at all in
Peoria
I lost those little town blues in
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas

4-2. Try using **?** as a delimiter. Change the current line

```
Las Vegas
```

to

Toledo

Because you are changing the whole line, you can also do this by using the change command, **c**.

4-3. Try searching backward in the file for the word

```
lost
```

and substitute

found

using the **?** as the delimiter. Did it work?

4-4. Search forward in the file for

no

and substitute

NO

for it. What happens if you try to use **?** as a delimiter?

Experiment with the various command combinations available for addressing a range of lines and doing global searches.

What happens if you try to substitute something for the **$$** ? Try to substitute **Big $** for $ on line 9 of your file. Type:

9s/$/Big $<CR>

What happened?

Special Characters

If you try to substitute the **$** sign in the line

```
I lost my $ in Las Vegas
```

you will find that instead of replacing the **$**, the new text is placed at the end of the line. The **$** is a special character in **ed** that is symbolic for the end of the line.

ed has several special characters that give you a shorthand for search patterns and substitution patterns. The characters act as wild cards. If you have tried to type in any of these characters, the result was probably different than what you had expected.

The special characters are:

.	Match any one character.
*****	Match zero or more occurrences of the preceding character.
.*	Match zero or more occurrences of any character following the period.
^	Match the beginning of the line.
$	Match the end of the line.
****	Take away the special meaning of the special character that follows.
&	Repeat the old text to be replaced in the new text of the replacement pattern.
[...]	Match the first occurrence of a character in the brackets.
[^...]	Match the first occurrence of a character that is NOT in the brackets.

In the following example, **ed** searches for any three-character sequence ending in the pattern at.

```
1,$p<CR>
rat
cat
turtle
cow
goat
g/.at<CR>
rat
cat
goat
```

Notice that the word goat is included because the string oat matches the string .at.

The * (asterisk) represents zero or more occurrences of a specified character in a search or substitute pattern. This can be useful in deleting repeated occurrences of a character that have been inserted by mistake. For example, suppose you hold down the R key too long while typing the word broke. You can use the * to delete every unnecessary R with one substitution command.

```
p<CR>
brrroke
s/br*/br<CR>
broke
```

Notice that the substitution pattern includes the b before the first r. If the b were not included in the search pattern, the * would interpret it, during the search, as a zero occurrence of r, make the substitution on it, and quit. (Remember, only the first occurrence of a pattern is changed in a substitution, unless you request a global search with **g**.) The following screen shows how the substitution would be made if you did not specify both the b and the r before the *.

```
p<CR>
brrroke
s/r*/r<CR>
rbrrroke
```

If you combine the period and the *, the combination will match all characters. With this combination you can replace all characters in the last part of a line:

```
p<CR>
Toads are slimy, cold creatures
s/are.*/are wonderful and warm<CR>
Toads are wonderful and warm
```

The .* can also replace all characters between two patterns.

```
p<CR>
Toads are slimy, cold creatures
s/are.*cre/are wonderful and warm cre<CR>
Toads are wonderful and warm creatures
```

If you want to insert a word at the beginning of a line, use the ^ (circumflex) for the old text to be substituted. This is very helpful when you want to insert the same pattern in the front of several lines. The next example places the word **all** at the beginning of each line:

```
1,$p<CR>
creatures great and small
things wise and wonderful
things bright and beautiful
1,$s/^/all /<CR>
1,$p<CR>
all creatures great and small
all things wise and wonderful
all things bright and beautiful
```

The $ sign is useful for adding characters at the end of a line or a range of lines:

```
1,$p<CR>
I love
I need
I use
The IRS wants my
1,$s/$/ money.<CR>
1,$p<CR>
I love money.
I need money.
I use money.
The IRS wants my money.
```

In these examples, you must remember to put a space after the word `all` or before the word `money` because **ed** adds the specified characters to the very beginning or the very end of the sentence. If you forget to leave a space before the word `money`, your file will look like this:

```
1,$s/$/money/<CR>
1,$p<CR>
I lovemoney
I needmoney
I usemoney
The IRS wants mymoney
```

The **$** sign also provides a handy way to add punctuation to the end of a line:

```
1,$p<CR>
I love money
I need money
I use money
The IRS wants my money
1,$s/$/./<CR>
1,$p/<CR>
I love money.
I need money.
I use money.
The IRS wants my money.
```

Because . is not matching a character (old text), but replacing a character (new text), it does not have a special meaning. To change a period in the middle of a line, you must take away the special meaning of the period in the old text. To do this, simply precede the period with a backslash (\). This is how you take away the special meaning of some special characters that you want to treat as normal text characters in search or substitute arguments. For example, the following screen shows how to take away the special meaning of the period:

```
p<CR>
Way to go.  Wow!
s/\./!<CR>
Way to go!  Wow!
```

The same method can be used with the backslash character itself. If you want to treat a \ as a normal text character, be sure to precede it with a \. For example, if you want to replace the \ symbol with the word backslash, use the substitute command line shown in the following screen:

```
1,2p<CR>
This chapter explains
how to use the \.
s/\\/backslash<CR>
how to use the backslash.
```

If you want to add text without changing the rest of the line, the & provides a useful shortcut. The & (ampersand) repeats the old text in the replacement pattern, so you do not have to type the pattern twice. For example:

```
p<CR>
The neanderthal skeletal remains
s/thal/& man's/<CR>
p<CR>
The neanderthal man's skeletal remains
```

ed automatically remembers the last string of characters in a search pattern or the old text in a substitution. However, you must prompt **ed** to repeat the replacement characters in a substitution with the % sign. The % sign allows you to make the same substitution on multiple lines without requesting a global

substitution. For example, to change the word money to the word gold, repeat the last substitution from line 1 on line 3, but not on line 4.

```
1,$n<CR>
1       I love money
2       I need food
3       I use money
4       The IRS wants my money
1s/money/gold<CR>
I love gold
3s//%<CR>
I use gold
1,$n<CR>
1       I love gold
2       I need food
3       I use gold
4       The IRS wants my money
```

ed automatically remembers the word `money` (the old text to be replaced), so that string does not have to be repeated between the first two delimiters. The % sign tells **ed** to use the last replacement pattern, **gold**.

ed tries to match the first occurrence of one of the characters enclosed in brackets and substitute the specified old text with new text. The brackets can be at any position in the pattern to be replaced.

In the following example, **ed** changes the first occurrence of the numbers `6`, `7`, `8`, or `9` to `4` on each line in which it finds one of those numbers:

```
1,$p<CR>
Monday      33,000
Tuesday     75,000
Wednesday   88,000
Thursday    62,000
1,$s/[6789]/4<CR>
Monday      33,000
Tuesday     45,000
Wednesday   48,000
Thursday    42,000
```

The next example deletes the Mr or Ms from a list of names:

```
1,$p<CR>
Mr Arthur Middleton
Mr Matt Lewis
Ms Anna Kelley
Ms M. L. Hodel
1,$s/M[rs] //<CR>
1,$p<CR>
Arthur Middleton
Matt Lewis
Anna Kelley
M. L. Hodel
```

If a ^ (circumflex) is the first character in brackets, **ed** interprets it as an instruction to match characters that are NOT within the brackets. However, if the circumflex is in any other position within the brackets, **ed** interprets it literally, as a circumflex.

```
1,$p<CR>
grade  A   Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  D   Jogging
grade  C   Tennis
1,$s/grade [^AB]/grade A<CR>
1,$p<CR>
grade  A   Computer Science
grade  B   Robot Design
grade  A   Boolean Algebra
grade  A   Jogging
grade  A   Tennis
```

Whenever you use special characters as wild cards in the text to be changed, remember to use a unique pattern of characters. In the above example, if you had used only

1,$s/[^AB]/A<CR>

you would have changed the g in the word grade to A. Try it.

Experiment with these special characters. Find out what happens (or does not happen) if you use them in different combinations.

Figure 5-7 summarizes the special characters for search or substitute patterns.

Command	Function
.	Match any one character in a search or substitute pattern.
*	Match zero or more occurrences of the preceding character in a search or substitute pattern.
.*	Match zero or more occurrences of any characters following the period.
^	Match the beginning of the line in the substitute pattern to be replaced or in a search pattern.
$	Match the end of the line in the substitute pattern to be replaced.
\	Take away the special meaning of the special character that follows in the substitute or search pattern.
&	Repeat the old text to be replaced in the new text replacement pattern.
%	Match the last replacement pattern.
[...]	Match the first occurrence of a character in the brackets.
[^...]	Match the first occurrence of a character that is NOT in the brackets.

Figure 5-7: Summary of Special Characters

Exercise 5

5-1. Create a file that contains the following lines of text.

A Computer Science
D Jogging
C Tennis

What happens if you try this command line:

1,$s/[^AB]/A/<CR>

Undo the above command. How can you make the C and D unique? (Hint: they are at the beginning of the line, in the position shown by the ^.) Do not be afraid to experiment!

5-2. Insert the following line above line 2:

These are not really my grades.

Using brackets and the ^ character, create a search pattern that you can use to locate the line you inserted. There are several ways to address a line. When you edit text, use the way that is quickest and easiest for you.

5-3. Add the following lines to your file:

I love money
I need money
The IRS wants my money

Now use one command to change them to:

It's my money
It's my money
The IRS wants my money

Using two command lines, do the following: change the word on the first line from **money** to **gold**, and change the last two lines from **money** to **gold** without using the words **money** or **gold** themselves.

5-4. How can you change the line

```
1020231020
```

to

10202031020

without repeating the old digits in the replacement pattern?

5-5. Create a line of text containing the following characters.

*** . \ & % ^ ***

Substitute a letter for each character. Do you need to use a backslash for every substitution?

Moving Text

You have now learned to address lines, create and delete text, and make substitutions. **ed** has one more set of versatile and important commands. You can move, copy, or join lines of text in the editing buffer. You can also read in text from a file that is not in the editing buffer, or write lines of the file in the buffer to another file in the current directory. The commands that move text are:

m	move lines of text
t	copy lines of text
j	join contiguous lines of text
w	write lines of text to a file
r	read in the contents of a file

Move Lines of Text

The **m** command allows you to move blocks of text to another place in the file. The general format is:

[*address1*,*address2*]**m**[*address3*] <**CR**>

The components of this command line include:

address1,*address2*
: The range of lines to be moved. If only one line is moved, only *address1* is given. If no address is given, the current line is moved.

m
: The move command.

address3
: Place the text after this line.

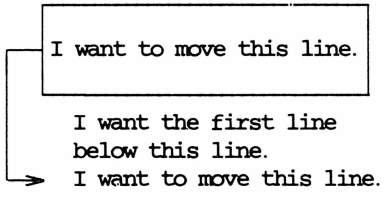
Try the following example to see how the command works. Create a file that contains these three lines of text:

I want to move this line.
I want the first line
below this line.

Type:

1m3<CR>

ed will move line 1 below line 3.



The next screen shows how this will appear on your terminal:

```
1,$p<CR>
I want to move this line.
I want the first line
below this line.
1m3<CR>
1,$p<CR>
I want the first line
below this line.
I want to move this line.
```

If you want to move a paragraph of text, have *address1* and *address2* define the range of lines of the paragraph.

In the following example, a block of text (lines 8 through 12) is moved below line 65. Notice the **n** command that prints the line numbers of the file:

```
8,12n<CR>
8               This is line 8.
9               It is the beginning of a
10              very short paragraph.
11              This paragraph ends
12              on this line.
64,65n<CR>
64              Move the block of text
65              below this line.
8,12m65<CR>
59,65n<CR>
59              Move the block of text
60              below this line.
61              This is line 8.
62              It is the beginning of a
63              very short paragraph.
64              This paragraph ends
65              on this line.
```

How can you move lines above the first line of the file? Try the following command.

3,4m0<CR>

When *address3* is 0, the lines are placed at the beginning of the file.

Copy Lines of Text

The copy command **t** (transfer) acts like the **m** command except that the block of text is not deleted at the original address of the line. A copy of that block of text is placed after a specified line of text. The general format of the command line is also similar.

The general format of the **t** command also looks like the **m** command.

[*address1*,*address2*]**t**[*address3*] <**CR**>

address1,*address2*
: The range of lines to be copied. If only one line is copied, only *address1* is given. If no address is given, the current line is copied.

t
: The copy command.

address3
: Place the copy of the text after this line.

The next example shows how to copy three lines of text below the last line.

```
                    Safety procedures:

          If there is a fire in the building:
          Close the door of the room to seal off the fire

          Break glass of nearest alarm.
          Pull lever.
          Locate and use fire extinguisher.

                          .
                          .
                          .
          A chemical fire in the lab requires that you:

          Break glass of nearest alarm
          Pull lever
          Locate and use fire extinguisher
```

The commands and **ed**'s responses to them are displayed in the next screen. Again, the **n** command displays the line numbers:

```
5,8n<CR>
5       Close the door of the room, to seal off the fire.
6       Break glass of nearest alarm.
7       Pull lever.
8       Locate and use fire extinguisher.
30n<CR>
30      A chemical fire in the lab requires that you:
6,8t30<CR>
30,$n<CR>
30      A chemical fire in the lab requires that you:
31      Break glass of nearest alarm
32      Pull lever
33      Locate and use fire extinguisher
6,8n<CR>
6       Break glass of nearest alarm
7       Pull lever
8       Locate and use fire extinguisher
```

The text in lines 6 through 8 remains in place. A copy of those three lines is placed after line 50.

Experiment with **m** and **t** on one of your files.

Joining Contiguous Lines

The **j** command joins the current line with the following line. The general format is:

[*address1,address1*]**j**<CR>

The next example shows how to join several lines together. An easy way of doing this is to display the lines you want to join using **p** or **n**.

```
1,2p<CR>
Now is the time to join
the team.
p<CR>
the team.
1p<CR>
Now is the time to join
j<CR>
p<CR>
Now is the time to jointhe team.
```

Notice that there is no space between the last word (`join`) and the first word of the next line (`the`), and the last word (`play`). You must place a space between them by using the **s** command.

Write Lines of Text to a File

The **w** command writes text from the buffer into a file. The general format is:

[*address1*,*address2*]**w** [*filename*]**<CR>**

address1,*address2*
: The range of lines to be placed in another file. If you do not use *address1* or *address2*, the entire file is written into a new file.

w
: The write command.

filename
: The name of the new file that contains a copy of the block of text.

In the following example the body of a letter is saved in a file called **memo**, so that it can be sent to other people.

```
1,$n<CR>
1                March 17, 1986
2          Dear Kelly,
3          There will be a meeting in the
4          green room at 4:30 P.M. today.
5          Refreshments will be served.
3,6w  memo<CR>
87
```

The **w** command places a copy of lines three through six into a new file called **memo**. **ed** responds with the number of characters in the new file.

Problems

The **w** command overwrites preexisting files; it erases the current file and puts the new block of text in the file without warning you. If, in our example, a file called **memo** had existed before we wrote our new file to that name, the original file would have been erased.

In "Special Commands," later in this chapter, you will learn how to execute shell commands from **ed**. Then you can list the file names in the directory to make sure that you are not overwriting a file.

Another potential problem is that you cannot write other lines to the file **memo**. If you try to add lines 13 through 16, the existing lines (3 through 6) will be erased and the file will contain only the new lines (13 through 16).

Read in the Contents of a File

The **r** command can be used to append text from a file to the buffer. The general format for the read command is:

[*address1*]**r** *filename* <**CR**>

address1 The text will be placed after the line *address1*. If *address1* is not given, the file is added to the end of the buffer.

r The read command.

filename The name of the file that will be copied into the editing buffer.

Using the example from the write command, the next screen shows a file being edited and new text being read into it.

```
1,$n<CR>
1                    March 17, 1986
2          Dear Michael,
3          Are you free later today?
4          Hope to see you there.
3r memo<CR>
87
3,$n<CR>
3          Are you free later today?
4          There is a meeting in the
5          green room at 4:30 P.M. today.
6          Refreshments will be served.
7          Hope to see you there.
```

ed responds to the read command with the number of characters in the file being added to the buffer (in the example, **memo**).

It is a good idea to display new or changed lines of text to be sure that they are correct.

Figure 5-8 summarizes the **ed** commands for moving text.

Command	Function
m	**move lines of text**
t	**copy lines of text**
j	**join contiguous lines**
w	**write text into a new file**
r	**read in text from another file**

Figure 5-8: Summary of **ed** Commands for Moving Text

Exercise 6

6-1. There are two ways to copy lines of text in the buffer: by issuing the copy command; or by using the write and read commands to first write text to a file and then read the file into the buffer.

Writing to a file and then reading the file into the buffer is a longer process. Can you think of an example where this method would be more practical?

What commands can you use to copy lines 10 through 17 of file **exer** into the file **exer6** at line 7?

6-2. Lines 33 through 46 give an example that you want placed after line 3, and not after line 32. What command performs this task?

6-3. Say you are on line 10 of a file and you want to join lines 13 and 14. What commands can you issue to do this?

Other Useful Commands and Information

There are four other commands and a special file that will be useful to you during editing sessions.

h,H	access the help commands, which provide error messages
l	display characters that are not normally displayed
f	display the current file name
!	temporarily escape **ed** to execute a shell command
ed.hup	When a system interrupt occurs, the **ed** buffer is saved in a special file named **ed.hup**.

Help Commands

You may have noticed when you were editing a file that **ed** responds to some of your commands with a ?. The ? is a diagnostic message issued by **ed** when it has found an error. The help commands give you a short message to explain the reason for the most recent diagnostic.

There are two help commands:

h Display a short error message that explains the reason for the most recent ?.

H Place **ed** into help mode so that a short error message is displayed every time the ? appears. (To cancel this request, type **H**.)

You know that if you try to quit **ed** without writing the changes in the buffer to a file, you will get a ?. Do this now. When the ? appears, type **h**:

```
q<CR>
?
h<CR>
warning: expecting 'w'
```

The ? is also displayed when you specify a new file name on the **ed** command line. Give **ed** a new file name. When the ? appears, type **h** to find out what the error message means.

```
ed newfile<CR>
? newfile
h<CR>
cannot open input file
```

This message means one of two things: either there is no file called **newfile** or there is such a file but **ed** is not allowed to read it.

As explained earlier, the **H** command responds to the ? and then turns on the help mode of **ed**, so that **ed** gives you a diagnostic explanation every time the ? is displayed subsequently. To turn off help mode, type **H** again. The next screen shows **H** being used to turn on help mode. Sample error messages are also displayed in response to some common mistakes:

```
$ ed newfile<CR>
e newfile<CR>
?newfile
H<CR>
cannot open input file
/hello<CR>
?
illegal suffix
1,22p<CR>
?
line out of range
a<CR>
I am appending this line to the buffer.
.<CR>
s/$ tea party<CR>
?
illegal or missing delimiter
,$s/$/ tea party<CR>
?
unknown command
H<CR>
q<CR>
?
h<CR>
warning: expecting 'w'
```

These are some of the most common error messages that you may encounter during editing sessions:

`illegal suffix`
: **ed** cannot find an occurrence of the search pattern **hello** because the buffer is empty.

`line out of range`
: **ed** cannot print any lines because the buffer is empty or the line specified is not in the buffer.

A line of text is appended to the buffer to show you some error messages associated with the **s** command.

`illegal or missing delimiter`
: The delimiter between the old text to be replaced and the new text is missing.

`unknown command`
: *address1* was not typed in before the comma; **ed** does not recognize **,$**.

Help mode is then turned off and **h** is used to determine the meaning of the last `?`. While you are learning **ed**, you may want to leave help mode turned on. If so, use the **H** command. However, once you become adept at using **ed**, you will only need to see error messages occasionally. Then you can use the **h** command.

Display Nonprinting Characters

If you are typing a tab character, the terminal will normally display up to eight spaces (covering the space up to the next tab setting. (Your tab setting may be more or less than eight spaces. See Chapter 7, "Shell Tutorial," on setting using **stty**).

If you want to see how many tabs you have inserted into your text, use the **l** (list) command. The general format for the **l** command is the same as for **n** and **p**.

[*address1*,*address2*]**l**<**CR**>

The components of this command line are:

address1,*address2*
: The range of lines to be displayed. If no address is given, the current line will be displayed. If only *address1* is given, only that line will be displayed.

l The command that displays the nonprinting characters along with the text.

The **l** command denotes tabs with a > (greater than) character. To type control characters, hold down the CONTROL key and press the appropriate alphabetic key. The key that sounds the bell is ^g (control-g). It is displayed as \07 which is the octal representation (the computer's code) for ^g.

Type in two lines of text that contain a **<^g>** (control-g) and a tab. Then use the **l** command to display the lines of text on your terminal.

```
a<CR>
Add a <^g> (control-g) to this line.<CR>
Add a <tab> (tab) to this line.<CR>
.<CR>
1,2l<CR>
Add a \07 (control-g) to this line.<CR>
Add a > (tab) to this line.<CR>
```

Did the bell sound when you typed **<^g>**?

The Current File Name

In a long editing session, you may forget the file name. The **f** command will remind you which file is currently in the buffer. Or, you may want to preserve the original file that you entered into the editing buffer and write the contents of the buffer to a new file. In a long editing session, you may forget, and accidentally overwrite the original file with the customary **w** and **q** command sequence. You can prevent this by telling the editor to associate the contents of the buffer with a new file name while you are in the middle of the editing session. This is done with the **f** command and a new file name.

The format for displaying the current file name is **f** alone on a line:

f<CR>

To see how **f** works, enter **ed** with a file. For example, if your file is called **oldfile**, **ed** will respond as shown in the following screen:

```
ed oldfile<CR>
323
f<CR>
oldfile
```

To associate the contents of the editing buffer with a new file name use this general format:

f newfile<CR>

If no file name is specified with the write command, **ed** remembers the file name given at the beginning of the editing session and writes to that file. If you do not want to overwrite the original file, you must either use a new file name with the write command, or change the current file name using the **f** command followed by the new file name. Because you can use **f** at any point in an editing session, you can change the file name immediately. You can then continue with the editing session without worrying about overwriting the original file.

The next screen shows the commands for entering the editor with **oldfile** and then changing its name to **newfile**. A line of text is added to the buffer and then the write and quit commands are issued.

```
ed oldfile<CR>
323
f<CR>
oldfile
f newfile<CR>
newfile
a<CR>
Add a line of text.<CR>
.<CR>
w<CR>
343
q<CR>
```

Once you have returned to the shell, you can list your files and verify the existence of the new file, **newfile**. **newfile** should contain a copy of the contents of **oldfile** plus the new line of text.

Escape to the Shell

How can you make sure you are not overwriting an existing file when you write the contents of the editor to a new file name? You need to return to the shell to list your files. The **!** allows you to temporarily return to the shell, execute a shell command, and then return to the current line of the editor.

The general format for the escape sequence is:

```
!shell command line<CR>
shell response to the command line
!
```

When you type the **!** as the first character on a line, the shell command must follow on that same line. The program's response to your command will appear as the command is running. When the command has finished executing, the **!** will be appear alone on a line. This means that that you are back in the editor at the current line.

For example, if you want to return to the shell to find out the correct date, type **!** and the shell command **date**.

```
p<CR>
This is the current line
! date<CR>
Tue  Apr 1  14:24:22  EST  1986
!
p<CR>
This is the current line.
```

The screen first displays the current line. Then the command is given to temporarily leave the editor and display the date. After the date is displayed, you are returned to the current line of the editor.

If you want to execute more than one command on the shell command line, see the discussion on **;** in the section called "Special Characters" in Chapter 7.

Recovering From System Interrupts

What happens if you are creating text in **ed** and there is an interrupt to the system, you are accidentally hung up on the system, or your terminal is unplugged? When an interrupt occurs, the UNIX system tries to save the contents of the editing buffer in a special file named **ed.hup**. Later you can retrieve your text from this file in one of two ways. First, you can use a shell command to move **ed.hup** to another file name, such as the name the file had while you were editing it (before the interrupt). Second, you can enter **ed** and use the **f** command to rename the contents of the buffer. An example of the second method is shown in the following screen:

```
ed ed.hup<CR>
928
f myfile<CR>
myfile
```

If you use the second method to recover the contents of the buffer, be sure to remove the **ed.hup** file afterward.

Conclusion

You now are familiar with many useful commands in **ed**. The commands that were not discussed in this tutorial, such as **G**, **P**, **Q** and the use of **( )** and { }, are discussed on the **ed**(1) page of the *User's Reference Manual*. You can experiment with these commands and try them to see what tasks they perform.

Figure 5-9 summarizes the functions of the commands introduced in this section.

Command	Function
h	Display a short error message for the preceding diagnostic ?.
H	Turn on help mode. An error message will be given with each diagnostic ?. The second **H** turns off help mode.
l	Display nonprinting characters in the text.
f	Display the current file name.
f *newfile*	Change the current file name associated with the editing buffer to *newfile*.
!cmd	Temporarily escape to the shell to execute a shell command *cmd*.
ed.hup	The editing buffer is saved in **ed.hup** if the terminal is hung up before a write command.

Figure 5-9: Summary of Other Useful Commands

Exercise 7

7-1. Create a new file called **newfile1**. Access **ed** and change the file's name to **current1**. Then create some text and write and quit **ed**. Run the **ls** command to verify that there is not a file called **newfile1** in your directory. If you do the shell command **ls**, you will see the directory does not contain a file called **newfile1**.

7-2. Create a file named **file1**. Append some lines of text to the file. Leave append mode but do not write the file. Turn off your terminal. Then turn on your terminal and log in again. Issue the **ls** command in the shell. Is there a new file called **ed.hup**? Place **ed.hup** in **ed**. How can you change the current file name to **file1**? Display the contents of the file. Are the lines the same lines you created before you turned off your terminal?

7-3. While you are in **ed**, temporarily escape to the shell and send a mail message to yourself.

Answers to Exercises

Exercise 1

1-1.

```
$ ed junk<CR>
? junk
a<CR>
Hello world.<CR>
.<CR>
w<CR>
12
q<CR>
$
```

1-2.

```
$ ed junk<CR>
12
1,$p<CR>
Hello world.<CR>
q<CR>
$
```

The system did not respond with the warning question mark because you did not make any changes to the buffer.

1-3.

```
$ ed junk<CR>
12
a<CR>
Wendy's horse came through the window.<CR>
.<CR>
1,$p<CR>
Hello world.
Wendy's horse came through the window.
q<CR>
?
w stuff<CR>
60
q<CR>
$
```

Exercise 2

2-1.

```
$ ed towns<CR>
? towns
a<CR>
My kind of town is<CR>
Chicago<CR>
Like being no where at all in<CR>
Toledo<CR>
I lost those little town blues in<CR>
New York<CR>
I lost my heart in<CR>
San Francisco<CR>
I lost $$ in<CR>
Las Vegas<CR>
.<CR>
w<CR>
164
```

2-2.

```
3<CR>
Like being no where at all in
```

2-3.

```
-2,3p<CR>
My kind of town is
Chicago
Like being no where at all in
Toledo
I lost those little town blues in
New York
```

2-4.

```
.= <CR>
6
6<CR>
New York
```

2-5.

```
$<CR>
Las Vegas
```

2-6.

```
?town<CR>
I lost those little town blues in
?<CR>
My kind of town is
```

2-7.

```
g/in<CR>
My kind of town is
Like being no where at all in
I lost those little town blues in
I lost my heart in
I lost $$ in

v/in<CR>
Chicago
Toledo
New York
San Francisco
Las Vegas
```

Exercise 3

3-1.

```
$ ed ex3<CR>
?ex3
i<CR>
?
q<CR>
```

The ? after the **i** means there is an error in the command. There is no current line before which text can be inserted.

3-2.

```
$ ed towns<CR>
164
.n<CR>
10          Las Vegas
3i<CR>
Illinois<CR>
.<CR>
.i<CR>
or<CR>
Naperville<CR>
.<CR>
$i<CR>
hotels in<CR>
1,$n<CR>
 1  my kind of town is
 2  Chicago
 3  or
 4  Naperville
 5  Illinois
 6  Like being no where at all in
 7  Toledo
 8  I lost those little town blues in
 9  New York
10  I lost my heart in
11  San Francisco
12  I lost $$ in
13  hotels in
14  Las Vegas
```

3-3.

```
1,5n<CR>
1  My kind of town is
2  Chicago
3  or
4  Naperville
5  Illinois
2,5c<CR>
London<CR>
.<CR>
1,3n<CR>
1  My kind of town is
2  London
3  Like being no where at all
```

3-4.

```
.<CR>
Like being no where at all
/Tol<CR>
Toledo
c<CR>
Peoria<CR>
.<CR>
.<CR>
Peoria
```

3-5.

```
.<CR>
/New Y/c<CR>
Iron City<CR>
.<CR>
.<CR>
Iron City
```

Your search string need not be the entire word or line. It only needs to be unique.

Exercise 4

4-1.

```
v/little town/s/town/city<CR>
My kind of city is
London
Like being no where at all in
Peoria
Iron City
I lost my heart in
San Francisco
I lost $$ in
hotels in
Las Vegas
```

The line

```
I lost those little town blues in
```

was not printed because it was NOT addressed by the **v** command.

4-2.

```
.<CR>
Las Vegas
s?Las Vegas?Toledo<CR>
Toledo
```

4-3.

```
?lost?s??found<CR>
I found $$ in
```

4-4.

```
/no?s??NO<CR>
?
/no/s//NO<CR>
Like being NO where at all in
```

You cannot mix delimiters such as / and **?** in a command line.

The substitution command on line 9 produced this output:

```
I found $$ inBig $
```

It did not work correctly because the $ sign is a special character in **ed**.

Exercise 5

5-1.

```
$ ed file1 <CR>
? file1
a <CR>
A  Computer Science <CR>
D  Jogging <CR>
C  Tennis <CR>
. <CR>
1,$s/[^AB]/A/ <CR>
1,$p <CR>
A  Computer Science
A  Jogging
A  Tennis
u <CR>
```

```
1,$s/^[^AB]/A <CR>
1,$p <CR>
A  Computer Science
A  Jogging
A  Tennis
```

5-2.

```
2i<CR>
These are not really my grades.<CR>
1,$p<CR>
A  Computer Science
These are not really my grades.
A  Tennis
A  Jogging
/^[^A]<CR>
These are not really my grades
?^[T]<CR>
These are not really my grades
```

5-3.

```
1,$p<CR>
I love money
I need money
The IRS wants my money
g/^I/s/I.*m /It's my m<CR>
It's my money
It's my money
```

```
/s/money/gold<CR>
It's my gold
2,$s//%<CR>
The IRS wants my gold
```

5-4.

```
s/10202/&0<CR>
10202031020
```

5-5.

```
a<CR>
* . \ & % ^ *<CR>
.<CR>
s/*/a<CR>
a . \ & % ^ *
s/*/b<CR>
a . \ & % ^ b
```

Because there were no preceding characters, * substituted for itself.

```
s/ \./c<CR>
a c \ & % ^ b
s/ \\/d<CR>
a c d & % ^ b
s/&/e<CR>
a c d e % ^ b
s/%/f<CR>
a c d e f ^ b
```

The & and % are only special characters in the replacement text.

```
s/ \^/g<CR>
a c d e f g b
```

Exercise 6

6-1. Any time you have lines of text that you may want to have repeated several times, it may be easier to write those lines to a file and read in the file at those points in the text.

If you want to copy the lines into another file you must write them to a file and then read that file into the buffer

containing the other file.

```
ed exer<CR>
725
10,17 w temp<CR>
210
q<CR>
ed exer6<CR>
305
7r temp<CR>
210
```

The file **temp** can be called any file name.

6-2.

```
33,46m3<CR>
```

6-3.

```
.= <CR>
10
13p<CR>
This is line 13.
j<CR>
.p<CR>
This is line 13.and line 14.
```

Remember that .= gives you the current line.

Exercise 7

7-1.

```
$ ed newfile1 <CR>
? newfile1
f current1 <CR>
current1
a <CR>
This is a line of text <CR>
Will it go into newfile1 <CR>
or into current1 <CR>
. <CR>
w <CR>
66
q <CR>
$ ls <CR>
bin
current1
```

7-2.

```
ed file1 <CR>
? file1
a <CR>
I am adding text to this file. <CR>
Will it show up in ed.hup? <CR>
. <CR>
```

Turn off your terminal.

Log in again.

```
ed ed.hup<CR>
58
f file1<CR>
file1
1,$p<CR>
I am adding text to this file.
Will it show up in ed.hup?
```

7-3.

```
$ ed file1<CR>
58
! mail mylogin<CR>
You will get mail when<CR>
you are done editing!<CR>
.<CR>
!
```

6. SCREEN EDITOR TUTORIAL

Chapter 6: Screen Editor Tutorial (vi)

Introduction 6-1
- Suggestions for Reading this Tutorial 6-3

Getting Started 6-4
- Setting the Terminal Configuration 6-4
- Changing Your Environment 6-5
- Setting the Automatic RETURN 6-6

Creating a File 6-8
- How to Create Text: the Append Mode 6-9
- How to Leave Append Mode 6-10

Editing Text: the Command Mode 6-11
- How to Move the Cursor 6-11
- Moving the Cursor to the Right or Left 6-13
- How to Delete Text 6-16
- How to Add Text 6-18

Quitting **vi** 6-19

Exercise 1 6-22

Moving the Cursor Around the Screen 6-23
- Positioning the Cursor on a Character 6-23
  - Moving the Cursor to the Beginning or End of a Line 6-24
  - Searching for a Character on a Line 6-26

Line Positioning 6-27
The Minus Sign Motion Command 6-27
The Plus Sign Motion Command 6-28
Word Positioning 6-28
Positioning the Cursor by Sentences 6-32
Positioning the Cursor by Paragraphs 6-34
Positioning in the Window 6-35

Positioning the Cursor in Undisplayed Text 6-41
Scrolling the Text 6-41
The Control-f Command 6-41
The Control-d Command 6-42
The Control-b Command 6-42
The Control-u Command 6-44
Go to a Specified Line 6-44
Line Numbers 6-44
Searching for a Pattern of Characters: the / and ? Commands 6-46

Exercise 2 6-53

Creating Text 6-55
Appending Text 6-55
Inserting Text 6-55
Opening a Line for Text 6-57

Exercise 3 6-60

Deleting Text 6-61
Undoing Entered Text in Text Input Mode 6-61
Undo the Last Command 6-63

Delete Commands in Command Mode 6-63
Deleting Words 6-64
Deleting Paragraphs 6-65
Deleting Lines 6-66
Deleting Text After the Cursor 6-66

Exercise 4 6-68

Modifying Text 6-69
Replacing Text 6-69
Substituting Text 6-70
Changing Text 6-72

Cutting And Pasting Text Electronically 6-77
Moving Text 6-77
Fixing Transposed Letters 6-78
Copying Text 6-78
Copying or Moving Text Using Registers 6-80

Exercise 5 6-82

Special Commands 6-83
Repeating the Last Command 6-83
Joining Two Lines 6-83
Clearing and Redrawing the Window 6-84
Changing Lower Case to Upper Case and Vice Versa 6-84

Using Line Editing Commands in **vi** 6-86
Temporarily Returning to the Shell: the **:sh** and **:!** Commands 6-86
Writing Text to a New File: the **:w** Command 6-87
Finding the Line Number 6-88

Deleting the Rest of the Buffer 6-89
Adding a File to the Buffer 6-89
Making Global Changes 6-90

Quitting **vi** 6-93

Special Options for **vi** 6-96
Recovering a File Lost by an Interrupt 6-96
Editing Multiple Files 6-96
Viewing a File 6-97

Exercise 6 6-99

Answers To Exercises 6-100
Exercise 1 6-100
Exercise 2 6-101
Exercise 3 6-103
Exercise 4 6-104
Exercise 5 6-105
Exercise 6 6-105

Introduction

This chapter is a tutorial on the screen editor, **vi** (short for visual editor). The **vi** editor is a powerful and sophisticated tool for creating and editing files. It is designed for use with a video display terminal which is used as a window through which you can view the text of a file. A few simple commands allow you to make changes to the text that are quickly reflected on the screen.

The **vi** editor displays from one to many lines of text. It allows you to move the cursor to any point on the screen or in the file (by specifying places such as the beginning or end of a word, line, sentence, paragraph, or file) and create, change, or delete text from that point. You can also use some line editor commands, such as the powerful global commands that allow you to change multiple occurrences of the same character string by issuing one command. To move through the file, you can scroll the text forward or backward, revealing the lines below or above the current window, as shown in Figure 6-1.

NOTE: Not all terminals have text scrolling capability; whether or not you can take advantage of **vi**'s scrolling feature depends on what type of terminal you have.

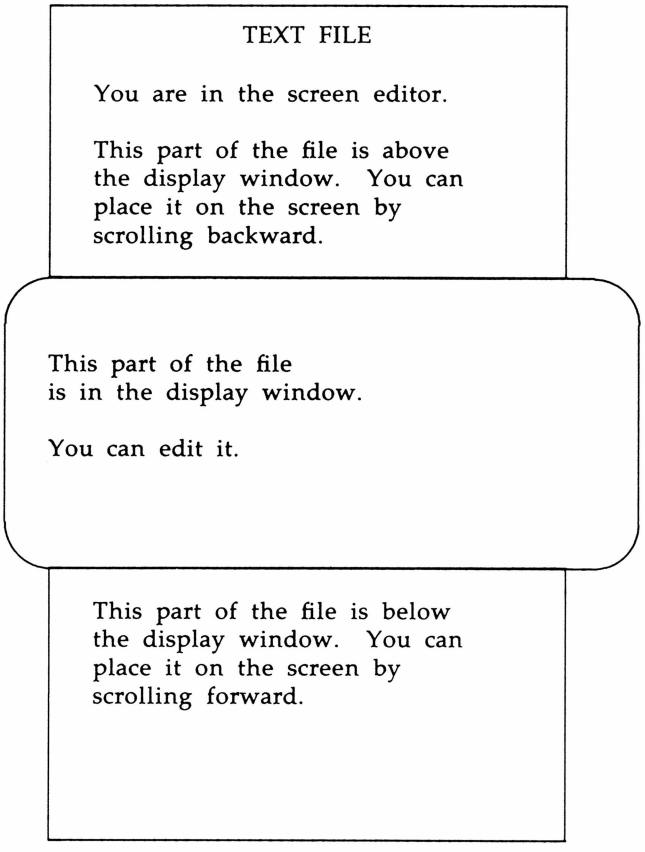


Figure 6-1: Displaying a File with a **vi** Window

There are more than 100 commands within vi. This chapter covers the basic commands that will enable you to use **vi** simply but effectively. Specifically, it explains how to do the following tasks:

- set up your terminal so that **vi** is accessible

- enter **vi**, create text, delete mistakes, write the text to a file, and quit
- move text within a file
- electronically cut and paste text
- use special commands and shortcuts
- temporarily escape to the shell to execute shell commands
- use line editing commands available within **vi**
- edit several files in the same session
- recover a file lost by an interruption to an editing session
- change your shell environment to set your terminal configuration and an automatic carriage return

Suggestions for Reading this Tutorial

As you read this tutorial, keep in mind the notation conventions described in the Preface. In the screens in this chapter arrows are also used to show the position of the cursor.

The commands discussed in each section are reviewed at the end of the section. A summary of **vi** commands is found in Appendix D, where they are listed by topic. At the end of some sections, exercises are given so you can experiment. The answers to all the exercises are at the end of this chapter. The best way to learn **vi** is by doing the examples and exercises as you read the tutorial. Log in on the UNIX system when you are ready to read this chapter.

Getting Started

The UNIX system is flexible; it can run on many types of computers and can be accessed from many kinds of terminals. However, because it is internally structured to be able to operate in so many ways, it needs to know what kind of hardware is being used in a given situation.

In addition, the UNIX system offers various optional features for using your terminal that you may or may not want to incorporate into your computing session routine. Your choice of these options, together with your hardware specifications, comprise your login environment. Once you have set up your login environment, the shell implements these specifications and options automatically every time you log in.

This section describes two parts of the login environment: setting the terminal configuration, which is essential for using **vi** properly, and setting the wrapmargin, or automatic (carriage) RETURN, which is optional.

Setting the Terminal Configuration

Before you enter **vi**, you must set your terminal configuration. This simply means that you tell the UNIX system what type of terminal you are using. This is necessary because the software for the **vi** editor is executed differently on different terminals.

Each type of terminal has several code names that are recognized by the UNIX system. Appendix F, "Setting Up the Terminal," tells you how to find a recognized name for your terminal. Keep in mind that many computer installations add terminal types to the list of terminals supported by default in your UNIX system. It is a good idea to check with your local system administrator for the most up-to-date list of available terminal types.

To set your terminal configuration, type

TERM = *terminal_name* **<CR>**
export TERM <CR>
tput init <CR>

The first line puts a value (a terminal type) in a variable called TERM. The second line exports this value; it conveys the value to all UNIX system programs whose execution depends on the type of terminal being used.

The **tput** command on the third line initializes (sets up) the software in your terminal so that it functions properly with the UNIX system. It is essential to run the **tput init** command when you are setting your terminal configuration because terminal functions such as tab settings will not work properly unless you do.

For example, if your terminal is a Teletype 5425 this is how your commands will appear on the screen.

```
$ TERM=5425<CR>
$ export TERM<CR>
$ tput init<CR>
```

Do not experiment by entering names for terminal types other than your terminal. This might confuse the UNIX system, and you may have to log off, hang up, or get help from your system administrator to restore your login environment.

Changing Your Environment

If you are going to use **vi** regularly, you should change your login environment permanently so you do not have to configure your terminal each time you log in. Your login environment is controlled by a file in your home directory called **.profile**. (This file, pronounced dot profile, does not exist in the file system; you must create it. For details, see Chapter 7.)

If you specify the setting for your terminal configuration in your **.profile**, your terminal will be configured automatically every time you log in. You can do this by adding the three lines shown in the last screen (the **TERM** assignment, **export** command, and **tput** command) to your **.profile**. (For detailed instructions, see Chapter 7.)

Setting the Automatic RETURN

NOTE: To set an automatic RETURN you must know how to create a file. If you are familiar with another text editor, such as **ed**, follow the instructions in this section. If you do not know how to use an editor but would like to have an automatic RETURN setting, skip this section for now and return to it when you have learned the basic skills taught in this chapter.

If you want the RETURN key to be entered automatically, create a file called **.exrc** in your home directory. You can use the **.exrc** file to contain options that control the **vi** editing environment. (For details about the **.exrc** file, see the *Editing Guide* or *Editing Utilities Guide*.)

To create a **.exrc** file, enter an editor with that file name. Then type in one line of text: a specification for the wrapmargin (automatic carriage return) option. The format for this option specification is

wm=*n*<**CR**>

n represents the number of characters from the righthand side of the screen where you want an automatic carriage return to occur. For example, say you want a carriage return at twenty characters from the righthand side of the screen. Type

wm=20<CR>

Finally, write the buffer contents to the file and quit the editor (see "Text Editing Buffers" in Chapter 4). The next time you log in, this

file will give you an automatic RETURN.

To check your settings for the terminal and wrapmargin when you are in **vi**, enter the command

:set<CR>

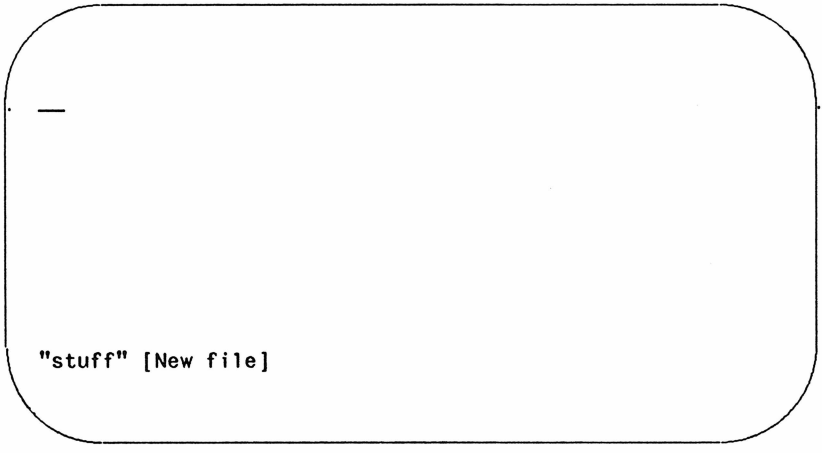
vi will report the terminal type and the wrapmargin, as well as any other options you may have specified. You can also use the **:set** command to create or change the wrapmargin option. Try experimenting with it.

Creating a File

First, enter the editor; type **vi** and the name of the file you want to create or edit.

vi *filename* **<CR>**

For example, say you want to create a file called **stuff**. When you type the **vi** command with the file name **stuff**, **vi** clears the screen and displays a window in which you can enter and edit text.



The _ (underscore) on the top line shows the cursor waiting for you to enter a command there. (On video display terminals the cursor may be a blinking underscore or a reverse color block.) Every other line is marked with a ~ (tilde), the symbol for an empty line.

If, before entering **vi**, you have forgotten to set your terminal configuration or have set it to the wrong type of terminal, you will see an error message instead.

```
$ vi stuff<CR>
terminal_name: unknown terminal type

[Using open mode]
"stuff" [New file]
```

You cannot set the terminal configuration while you are in the editor; you must be in the shell. Leave the editor by typing

: q<CR>

Then set the correct terminal configuration.

How to Create Text: the Append Mode

If you have successfully entered **vi**, you are in command mode and **vi** is waiting for your commands. How do you create text?

- Press the A key (**<a>**) to enter the append mode of **vi**. (Do not press the RETURN key.) You can now add text to the file. (An A is not printed on the screen.)
- Type in some text.
- To begin a new line, press the RETURN key.

  If you have specified the wrapmargin option in a **.exrc** file, you will get a new line whenever you get an automatic RETURN (see "Setting the Automatic RETURN").

How to Leave Append Mode

When you finish creating text, press the ESCAPE key to leave append mode and return to command mode. Then you can edit any text you have created or write the text in the buffer to a file.

```
<a>Create some text<CR>
in the screen editor<CR>
and return to<CR>
command mode.<ESC>
```

If you press the ESCAPE key and a bell sounds, you are already in command mode. The text in the file is not affected by this, even if you press the ESCAPE key several times.

Editing Text: the Command Mode

To edit an existing file you must be able to add, change, and delete text. However, before you can perform those tasks you must be able to move to the part of the file you want to edit. **vi** offers an array of commands for moving from page to page, between lines, and between specified points inside a line. These commands, along with commands for deleting and adding text, are introduced in this section.

How to Move the Cursor

To edit your text, you need to move the cursor to the point on the screen where you will begin the correction. This is easily done with four keys that are grouped together on the keyboard: h, j, k, and l.

<h> moves the cursor one character to the left

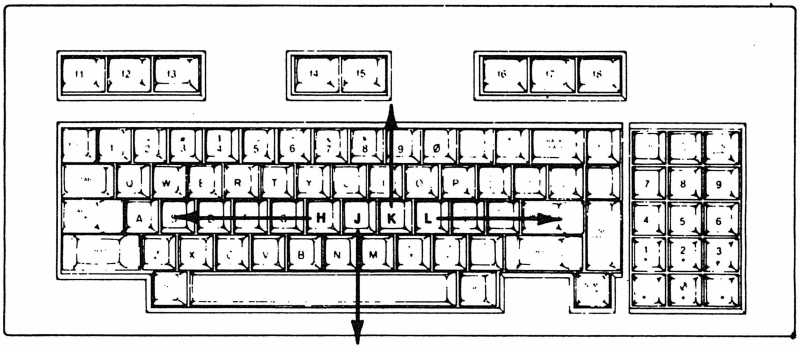
<j> moves the cursor down one line

<k> moves the cursor up one line

<l> moves the cursor one character to the right

The **<j>** and **<k>** commands maintain the column position of the cursor. For example, if the cursor is on the seventh character from the left, when you type **<j>** or **<k>** it goes to the seventh character on the new line. If there is no seventh character on the new line, the cursor moves to the last character.

Many people who use **vi** find it helpful to mark these four keys with arrows showing the direction in which each key moves the cursor.



NOTE: Some terminals have special cursor control keys that are marked with arrows. Use them in the same way you use the **<h>**, **<j>**, **<k>**, and **<l>** commands.

Watch the cursor on the screen while you press the keys **<h>**, **<j>**, **<k>**, and **<l>**. Instead of pressing a motion command key a number of times to move the cursor a corresponding number of spaces or lines, you can precede the command with the desired number. For example, to move two spaces to the right, you can press **<l>** twice or enter **<2l>**. To move up four lines, press **<k>** four times or enter **<4k>**. If you cannot go any farther in the direction you have requested, **vi** will sound a bell.

Now experiment with the **j** and **k** motion commands. First, move the cursor up seven lines. Type

<7k>

The cursor will move up seven lines above the current line. If there are less than seven lines above the current line, a bell will sound and the cursor will remain on the current line.

Now move the cursor down thirty-five lines. Type

<35j>

vi will clear and redraw the screen. The cursor will be on the thirty-fifth line below the current line, appearing in the middle of the new window. If there are less than thirty-five lines below the current line, the bell will sound and the cursor will remain on the current line. Watch what happens when you type the next command.

<35k>

Like most **vi** commands, the **<h>**, **<j>**, **<k>**, and **<l>** motion commands are silent; they do not appear on the screen as you enter them. The only time you should see characters on the screen is when you are in append mode and are adding text to your file. If the motion command letters appear on the screen, you are still in append mode. Press the ESCAPE key to return to command mode and try the commands again.

Moving the Cursor to the Right or Left

In addition to the motion command keys **<h>** and **<l>**, the space bar and the BACKSPACE key can be used to move the cursor right or left to a character on the current line.

<space bar>	move the cursor one character to the right
<*n*space bar>	move the cursor *n* characters to the right
<**BACKSPACE**>	move the cursor one character to the left

<*n*BACKSPACE> move the cursor *n* characters to the left

Try typing in a number before the command key. Notice that the cursor moves the specified number of characters to the left or right. In the example below, the cursor movement is shown by the arrows.

To move the cursor quickly to the right or left, prefix a number to the command. For example, suppose you want to create four columns in your screen. After you've finished typing the headings for the first three columns, you notice a typing mistake.

```
Column 1      Column 2      column

                          <ESC>
```

You want to correct your mistake before continuing. Exit insert mode and return to command mode by pressing the ESCAPE key; the cursor will move to the **n**. Then use the <**h**> command to move back five spaces.

```
Column 1      Column 2      column

                                <5h>
Column 1      Column 2      column

                    <x><i>C<ESC>
```

Erase the c by typing **<x>**. Then change to insert mode (**<i>**), enter a **C**, followed by pressing the ESCAPE key. Use the **<l>** motion command to return to your earlier position.

```
Column 1      Column 2       Column

                            <5l>
Column 1      Column 2       Column
```

By now you may have discovered that you can move the cursor back and forth on a line by using the space bar and the BACKSPACE key.

<space bar>	move the cursor one character to the right
<*n*space bar>	move the cursor *n* characters to the right
<BACKSPACE>	move the cursor one character to the left
<*n*BACKSPACE>	move the cursor *n* characters to the left

Again, you can specify a multiple space movement by typing a number before pressing the space bar or BACKSPACE key. The cursor will move the number of characters you request to the left or right.

How to Delete Text

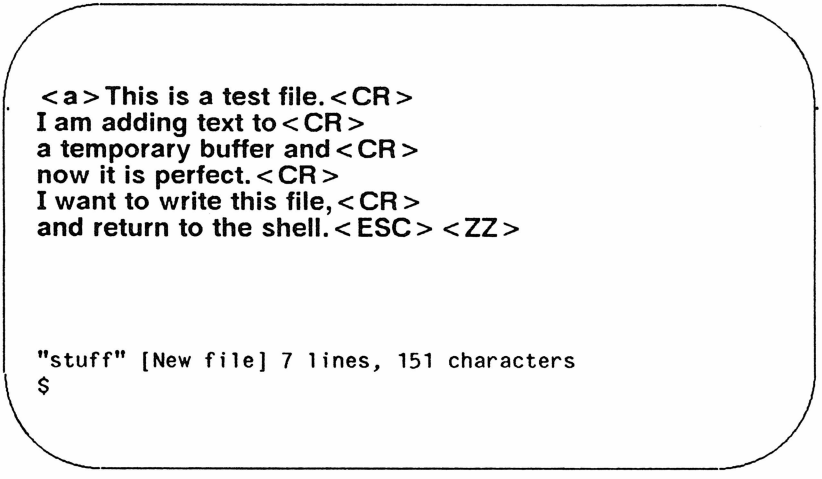
If you want to delete a character, move the cursor to that character and press the **<x>**. Watch the screen as you do so; the character will disappear and the line will readjust to the change. To erase three characters in a row, press **<x>** three times. In the following example, the arrows under the letters show the positions of the cursor.

<x>	delete one character
<*n*x>	delete *n* characters, where *n* is the number of characters you want to delete

```
Hello wurld!

      <x>

Hello wrld!
```

Quitting vi

When you have finished your text, you will want to write the buffer contents to a file and return to the shell. To do this, hold down the SHIFT key and press **Z** twice (**<ZZ>**). The editor remembers the file name you specified with the **vi** command at the beginning of the editing session, and moves the buffer text to the file of that name. A notice at the bottom of the screen gives the file name and the number of lines and characters in the file. Then the shell gives you a prompt.



You can also use the **:w** and **:q** commands of the line editor for writing and quitting a file. (Line editor commands begin with a colon and appear on the bottom line of the screen.) The **:w** command writes the buffer to a file. The **:q** command leaves the editor and returns you to the shell. You can type these commands separately or combine them into the single command **:wq**. It is easier to combine them.

```
<a>This is a test file.<CR>
I am adding text to<CR>
a temporary buffer and<CR>
now it is perfect.<CR>
I want to write this file,<CR>
and return to the shell.<ESC>



:wq<CR>
```

Figure 6-2 summarizes the basic commands you need to enter and use **vi**.

Command	Function
TERM = *terminal_name* **export TERM**	set the terminal configuration
tput init	initialize the terminal as defined by *terminal_name*
vi *filename*	enter **vi** editor to edit the file called *filename*
<a>	add text after the cursor
<h>	move one character to the left
<j>	move down one line
<k>	move up one line
<l>	move one character to the right
<x>	delete a character
<CR>	carriage return
<ESC>	leave append mode, and return to **vi** command mode
:w	write to a file
:q	quit **vi**
:wq	write to a file and quit **vi**
<ZZ>	write to a file and quit **vi**

Figure 6-2: Summary of Commands for the **vi** Editor

Exercise 1

Answers to the exercises are given at the end of this chapter. However, keep in mind that there is often more than one way to perform a task in **vi**. If your method works, it is correct.

As you give commands in the following exercises, watch the screen to see how it changes or how the cursor moves.

1-1. If you have not logged in yet, do so now. Then set your terminal configuration.

1-2. Enter **vi** and append the following five lines of text to a new file called **exer1**.

This is an exercise!
Up, down,
left, right,
build your terminal's
muscles bit by bit

1-3. Move the cursor to the first line of the file and the seventh character from the right. Notice that as you move up the file, the cursor moves in to the last letter of the file, but it does not move out to the last letter of the next line.

1-4. Delete the seventh and eighth characters from the right.

1-5. Move the cursor to the last character on the last line of the text.

1-6. Append the following new line of text:

and byte by byte

1-7. Write the buffer to a file and quit **vi**.

1-8. Reenter **vi** and append two more lines of text to the file **exer1**.
What does the notice at the bottom of the screen say once you have reentered **vi** to edit **exer1**?

Moving the Cursor Around the Screen

Until now you have been moving the cursor with the **<h>**, **<j>**, **<k>**, **<l>**, BACKSPACE key, and the space bar. There are several other commands that can help you move the cursor quickly around the screen. This section explains how to position the cursor in the following ways:

- by characters on a line
- by lines
- by text objects
  - words
  - sentences
  - paragraphs
- in the window

There are also commands that position the cursor within parts of the **vi** editing buffer that are not visible on the screen. These commands will be discussed in the next section, "Positioning the Cursor in Undisplayed Text."

To follow this section of the tutorial, you should enter **vi** with a file that contains at least forty lines. If you do not have a file of that length, create one now. Remember, to execute the commands described here, you must be in command mode of **vi**. Press the ESCAPE key to make sure that you are in command mode rather than append mode.

Positioning the Cursor on a Character

There are three ways to position the cursor on a character in a line.

- by moving the cursor right or left to a character

- by specifying the character at either end of the line
- by searching for a character on a line

The first method was discussed earlier in this chapter under "Moving the Cursor to the Right or Left." The following sections describe the other two methods.

Moving the Cursor to the Beginning or End of a Line

The second method of positioning the cursor on a line is by using one of three commands that put the cursor on the first or last character of a line.

<$>	puts the cursor on the last character of a line
<0> (zero)	puts the cursor on the first character of a line
<^> (circumflex)	puts the cursor on the first nonblank character of a line

The following examples show the movement of the cursor produced by each of these three commands.

```
Go to the end of the line!

          <$>

Go to the end of the line!
```

```
Go to the beginning of the line!


                              <0>
Go to the beginning of the line!
```

```
Go to the first character
of the line
     that is not blank!


                         <^>
Go to the first character
of the line
     that is not blank!
```

Searching for a Character on a Line

The third way to position the cursor on a line is to search for a specific character on the current line. If the character is not found on the current line, a bell sounds and the cursor does not move. (There is also a command that searches a file for patterns. This will be discussed in the next section.) There are six commands you can use to search within a line: **<f>**, **<F>**, **<t>**, **<T>**, **<;>**, and **<,>**. You must specify a character after all of them except the **<;>** and **<,>** commands.

<f*x***>** Move the cursor to the right to the specified character *x*.

<F*x***>** Move the cursor to the left to the specified character *x*.

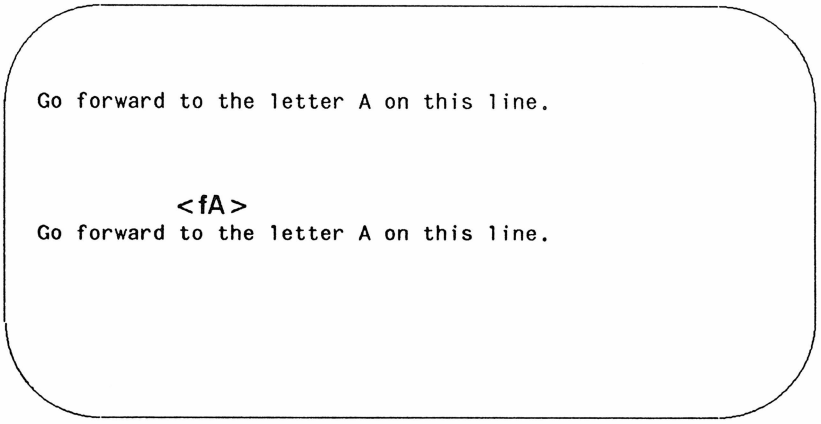
<t*x***>** Move the cursor right to the character just before the specified character *x*.

<T*x***>** Move the cursor left to the character just after the specified character *x*.

<;> Continue the search specified in the last command, in the same direction. The **;** remembers the character and seeks out the next occurrence of that character on the current line.

<,> Continue the search specified in the last command, in the opposite direction. The **,** remembers the character and seeks out the previous occurrence of that character on the current line.

For example, in the following screen **vi** searches to the right for the first occurrence of the letter A on the current line.



Try the search commands on one of your files.

Line Positioning

Besides the **<j>** and **<k>** commands that you have already used, the **< + >**, **<->**, and **<CR>** commands can be used to move the cursor to other lines.

The Minus Sign Motion Command

The **<->** command moves the cursor up a line, positioning it at the first nonblank character on the line. To move more than one line at a time, specify the number of lines you want to move before the **<->** command. For example, to move the cursor up thirteen lines, type:

<13– >

The cursor will move up thirteen lines. If some of those lines are above the current window, the window will scroll up to reveal them. This is a rapid way to move quickly up a file.

Now try to move up 100 lines. Type:

<100– >

What happened to the window? If there are less then 100 lines above the current line a bell will sound, telling you that you have

made a mistake, and the cursor will remain on the current line.

The Plus Sign Motion Command

The plus sign command (< + >) or the <**CR**> command moves the cursor down a line. Specify the number of lines you want to move before the < + > command. For example, to move the cursor down nine lines, type:

<**9+**>

The cursor will move down nine lines. If some of those lines are below the current screen, the window will scroll down to reveal them.

Now try to do the same thing by pressing the RETURN key. Were the results the same as when you pressed the key?

Word Positioning

The **vi** editor considers a word to be a string of characters that may include letters, numbers, or underscores. There are six word positioning commands: <**w**>, <**b**>, <**e**>, <**W**>, <**B**>, and <**E**>. The lower case commands (<**w**>, <**b**>, and <**e**>) treat any character other than a letter, digit, or underscore as a delimiter, signifying the beginning or end of a word. Punctuation before or after a blank is considered a word. The beginning or end of a line is also a delimiter.

The upper case commands (<**W**>, <**B**>, and <**E**>) treat punctuation as part of the word; words are delimited by blanks and newlines only.

The following is a summary of the word positioning commands.

<**w**> Move the cursor forward to the first character in the next word. You may press <**w**> as many times as you want to reach the word you want, or you can prefix the necessary number to the <**w**>.

<*n***w**> Move the cursor forward *n* number of words to the first character of that word. The end of the line does not stop the movement of the cursor; instead, the cursor

wraps around and continues counting words from the beginning of the next line.

```
The <w> command
leaps word by word through the
file.  Move from THIS word forward


          <6w>

six words to THIS word.
```

```
The <w> command
leaps word by word through the
file.  Move from THIS word forward
six words to THIS word.
```

<W> Ignore all punctuation and move the cursor forward to the word after the next blank.

<e> Moves the cursor forward in the line to the last character in the next word.

Go forward one word to the end of
the next word in this line

<e>

Go forward one word to the end of
the next word in this line

Go to the end of the third word after the current word.

<3e>

```
Go to the end of the third word after the current word.
```

<E> Ignores all punctuation except blanks, delimiting words only by blanks.

<b> Move the cursor backward in the line to the first character of the previous word.

<*n*b> Move the cursor backward *n* number of words to the first character of the *n*th word. The **<b>** command does not stop at the beginning of a line, but moves to the end of the line above and continues moving backward.

<B> Can be used just like the **<b>** command, except that it delimits the word only by blank spaces and newlines. It treats all other punctuation as letters of a word.

```
Leap backward word by word through
the file. Go back four words from here.

                    <4b>
```

```
the file. Go back four words from here.
```

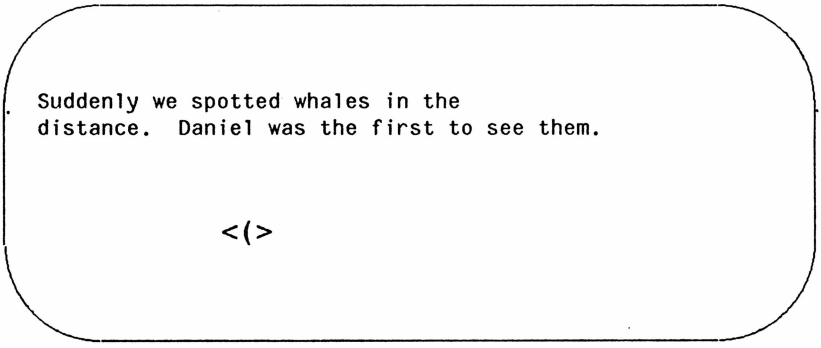
Positioning the Cursor by Sentences

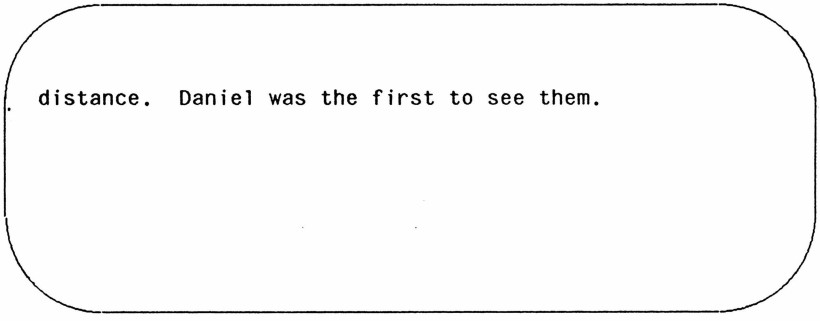
The **vi** editor also recognizes sentences. In **vi** a sentence ends in ! or . or ?. If these delimiters appear in the middle of a line, they must be followed by two blanks for **vi** to recognize them. You should get used to the **vi** convention of recognizing two blanks after a period as the end of a sentence, because it is often useful to be able to operate on a sentence as a unit.

You can move the cursor from sentence to sentence in the file with the <(> (open parenthesis) and <)> (close parenthesis) commands.

- < (> Move the cursor to the beginning of the current sentence.
- < *n*(> Move the cursor to the beginning of the *n*th sentence above the current sentence.
- <) > Move the cursor to the beginning of the next sentence.
- < *n*) > Move the cursor to the beginning of the *n*th sentence below the current sentence.

The example in the following screens shows how the open parenthesis moves the cursor around the screen.





Now repeat the command, preceding it with a number. For example, type:

<3(> (or)
<5)>

Did the cursor move the correct number of sentences?

Positioning the Cursor by Paragraphs

Paragraphs are recognized by **vi** if they begin after a blank line. If you want to be able to move the cursor to the beginning of a paragraph (or later in this tutorial, to delete or change a whole paragraph), then make sure each paragraph ends in a blank line.

<{>	Move the cursor to the beginning of the current paragraph, which is delimited by a blank line above it.
<**n**{>	Move the cursor to the beginning of the *n*th paragraph above the current paragraph.
<}>	Move the cursor to the beginning of the next paragraph.
<**n**}>	Move the cursor to the *n*th paragraph below the current line.

The following two screens show how the cursor can be moved to the beginning of another paragraph.

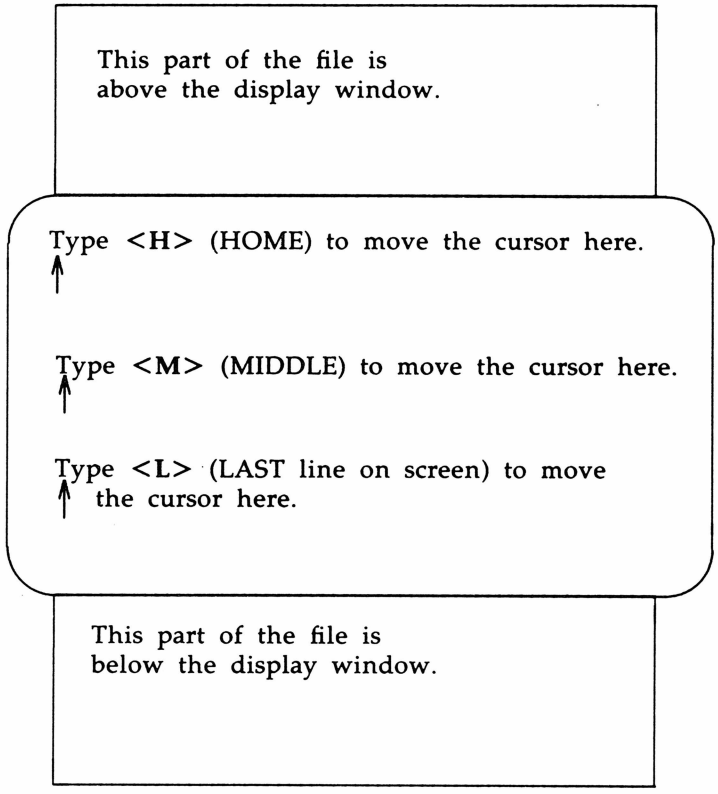
```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.



          <}>
"Hey look!  Here come the whales!" he cried excitedly.
```

```
Suddenly, we spotted whales in the


distance.  Daniel was the first to see them.

"Hey look!  Here come the whales!" he cried excitedly.
```

Positioning in the Window

The **vi** editor also provides three commands that help you position yourself in the window. Try out each command. Be sure to type them in upper case.

<H>	Move the cursor to the first line on the screen.
<M>	Move the cursor to the middle line on the screen.
<L>	Move the cursor to the last line on the screen.



Figures 6-3 through 6-6 summarize the **vi** commands for moving the cursor by positioning it on a character, line, word, sentence, paragraph, or position on the screen. (Additional **vi** commands for moving the cursor are summarized in Figure 6-7, later in the chapter.)

Positioning on a Character	
<h>	Move the cursor one character to the left.
<l>	Move the cursor one character to the right.
<BACKSPACE>	Move the cursor one character to the left.
<space bar>	Move the cursor one character to the right.
<f*x*>	Move the cursor to the right to the specified character *x*.
<F*x*>	Move the cursor to the left to the specified character *x*.
<t*x*>	Move the cursor to the right, to the character just before the specified character *x*.
<T*x*>	Move the cursor to the left, to the character just after the specified character *x*.
<;>	Continue searching in same direction on the line for the last character requested with **<f>**, **<F>**, **<t>**, or **<T>**. The **;** remembers the character and finds the next occurrence of it on the current line.
<,>	Continue searching in opposite direction on the line for the last character requested with **<f>**, **<F>**, **<t>**, or **<T>**. The **,** remembers the character and finds the next occurrence of it on the current line.

Figure 6-3: Summary of **vi** Motion Commands (Sheet 1 of 4)

Positioning on a Line	
<k>	Move the cursor up to the same column in the previous line (if a character exists in that column).
<j>	Move the cursor down to the same column in the next line (if a character exists in that column).
<->	Move the cursor up to the beginning of the previous line.
< >	Move the cursor down to the beginning of the next line.
<CR>	Move the cursor down to the beginning of the next line.

Figure 6-4: Summary of **vi** Motion Commands (Sheet 2 of 4)

Positioning on a Word	
<w>	Move the cursor forward to the first character in the next word.
<W>	Ignore all punctuation and move the cursor forward to the next word delimited only by blanks.
<b>	Move the cursor backward one word to the first character of that word.
<B>	Move the cursor to the left one word, which is delimited only by blanks.
<e>	Move the cursor to the end of the current word.
<E>	Delimit the words by blanks only. The cursor is placed on the last character before the next blank space, or end of the line.

Figure 6-5: Summary of **vi** Motion Commands (Sheet 3 of 4)

Positioning on a Sentence	
<(>	Move the cursor to the beginning of the current sentence.
<) >	Move the cursor to the beginning of the next sentence.
Positioning on a Paragraph	
<{ >	Move the cursor to the beginning of the current paragraph.
<} >	Move the cursor to the beginning of the next paragraph.
Positioning in the Window	
<H >	Move the cursor to the first line on the screen (the home position).
<M >	Move the cursor to the middle line on the screen.
<L >	Move the cursor to the last line on the screen.

Figure 6-6: Summary of **vi** Motion Commands (Sheet 4 of 4)

Positioning the Cursor in Undisplayed Text

How do you move the cursor to text that is not shown in the current editing window? One option is to use the **<20j>** or **<20k>** command. However, if you are editing a large file, you need to move quickly and accurately to another place in the file. This section covers those commands that can help you move around within the file in the following ways:

- by scrolling forward or backward in the file
- by going to a specified line in the file
- by searching for a pattern in the file

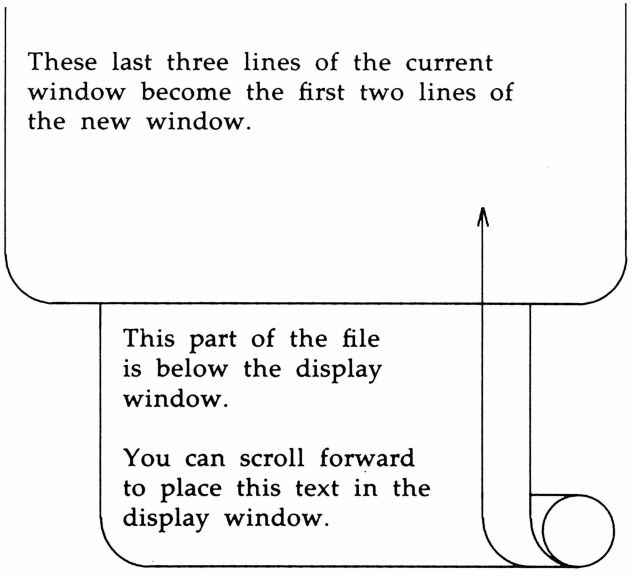
Scrolling the Text

Four commands allow you to scroll the text of a file. The **<^f>** (control-f) and **<^d>** (control-d) commands scroll the screen forward. The **<^b>** (control-b) and **<^u>** (control-u) commands scroll the screen backward.

The Control-f Command

The **<^f>** (control-f) command scrolls the text forward one full window of text below the current window. To do this **vi** clears the screen and redraws the window. The three lines that were at the bottom of the current window are placed at the top of the new window. If there are not enough lines left in the file to fill the window, the screen displays a ~ (tilde) to show that there are empty lines.

vi clears and redraws the screen as follows:

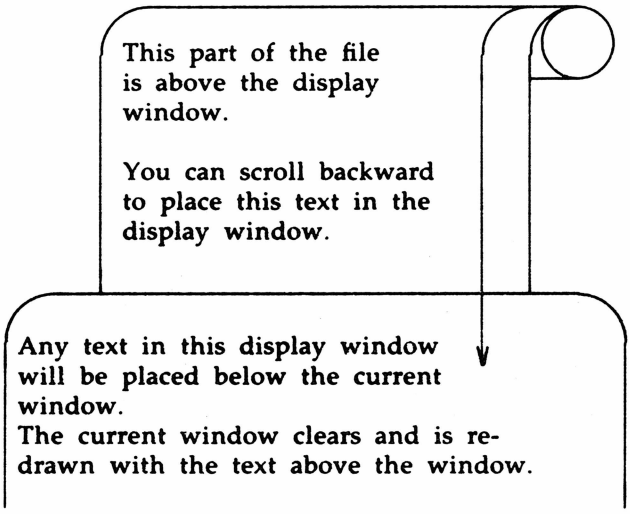


The Control-d Command

The <^**d**> (control-d) command scrolls down a half screen to reveal text below the window. When you type <^**d**>, the text appears to be rolled up at the top and unrolled at the bottom. This allows the lines below the screen to appear on the screen, while the lines at the top of the screen disappear. If there are not enough lines in the file, a bell will sound.

The Control-b Command

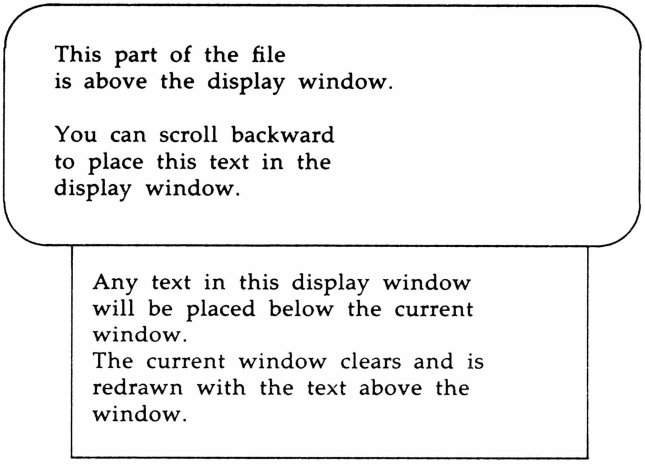
The <^**b**> (control-b) command scrolls the screen back a full window to reveal the text above the current window. To do this, **vi** clears the screen and redraws the window with the text that is above the current screen. Unlike the <^**f**> command, <^**b**> does not leave any reference lines from the previous window. If there are not enough lines above the current window to fill a full new window, a bell will sound and the current window will remain on the screen.



Now try scrolling backward. Type

<^b>

vi clears the screen and draws a new screen.



Any text that was in the display window is placed below the current window.

The Control-u Command

The <^u> (control-u) command scrolls up a half screen of text to reveal the lines just above the window. The lines at the bottom of the window are erased. Now scroll down in the text, moving the portion below the screen into the window. Type:

<^u>

When the cursor reaches the top of the file, a bell sounds to notify you that the file cannot scroll further.

Go to a Specified Line

The <G> command positions the cursor on a specified line in the window; if that line is not currently on the screen, <G> clears the screen and redraws the window around it. If you do not specify a line, <G> goes to the last line of the file.

<G> go to the last line of the file

<*n*G> go to the *n*th line of the file

Line Numbers

Each line of the file has a line number corresponding to its position in the buffer. To get the number of a particular line, position the cursor on it and type <^g>. The <^g> command gives you a status notice at the bottom of the screen which tells you:

- the name of the file
- if the file has been modified
- the line number on which the cursor rests
- the total number of lines in the buffer
- the percentage of the total lines in the buffer represented by the current line

```
This line is the 35th line of the buffer.
The cursor is on this line.


                    <^g>

There are several more lines in the
buffer.
The last line of the buffer is line 116.
```

```
This line is the 35th line of the buffer.
The cursor is on this line.


There are several more lines in the
buffer.
The last line of the buffer is line 116.

"file.name" [modified] line 36 of 116 --34%--
```

Searching for a Pattern of Characters: the / and ? Commands

The fastest way to reach a specific place in your text is by using one of the search commands: /, **?**, **<n>**, or **<N>**. These commands allow you to search forward or backward in the buffer for the next occurrence of a specified character pattern. The / and **?** commands are not silent; they appear as you type them, along with the search pattern, on the bottom of the screen. The **<n>** and **<N>** commands, which allow you to repeat the requests you made for a search with a / or **?** command, are silent.

The /, followed by a *pattern* (*/pattern*), searches forward in the buffer for the next occurrence of the characters in *pattern*, and puts the cursor on the first of those characters. For example, the command line

/Hello world<CR>

finds the next occurrence in the buffer of the words **Hello world** and puts the cursor under the **H**.

The **?**, followed by a *pattern* (*?pattern*), searches backward in the buffer for the first occurrence of the characters in *pattern*, and puts the cursor on the first of those characters. For example, the command line

?data set design<CR>

finds the last occurrence in the buffer (before your current position) of the words **data set design** and puts the cursor under the **d** in **data**.

These search commands do not wrap around the end of a line while searching for two words. For example, say you are searching for the words `Hello world`. If `Hello` is at the end of one line and `world` is at the beginning of the next, the search command will not find that occurrence of `Hello World`.

However, they do wrap around the end or the beginning of the buffer to continue a search. For example, if you are near the end of the buffer, and the pattern for which you are searching (with the */pattern* command) is at the top of the buffer, the command will find the pattern.

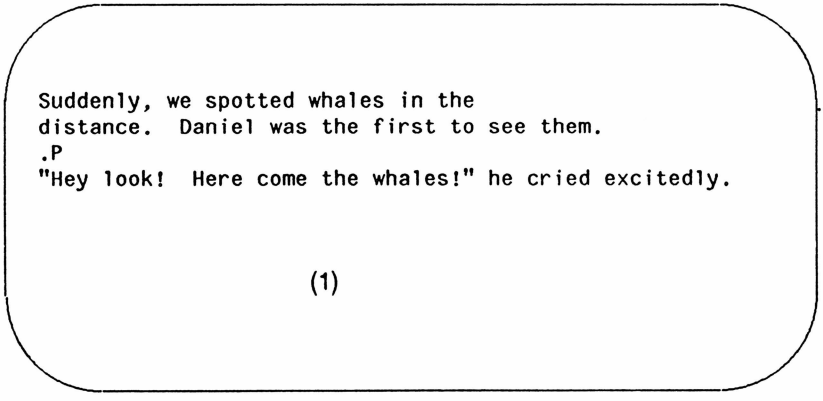
The **<n>** and **<N>** commands allow you to continue searches you have requested with */pattern* or **?***pattern* without retyping them.

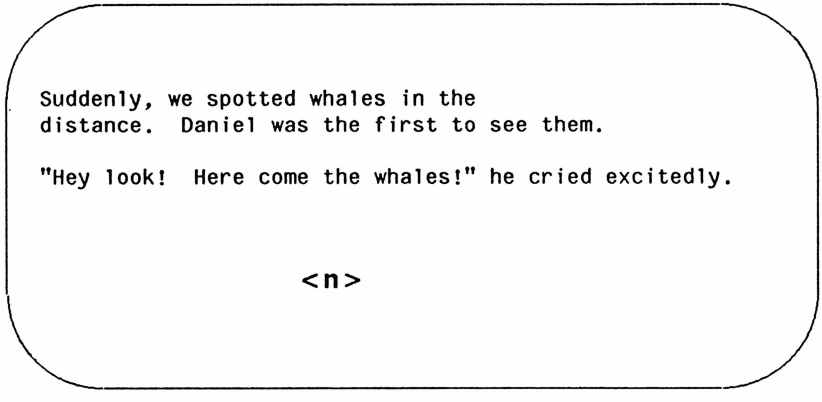
<n> Repeat the last search command.

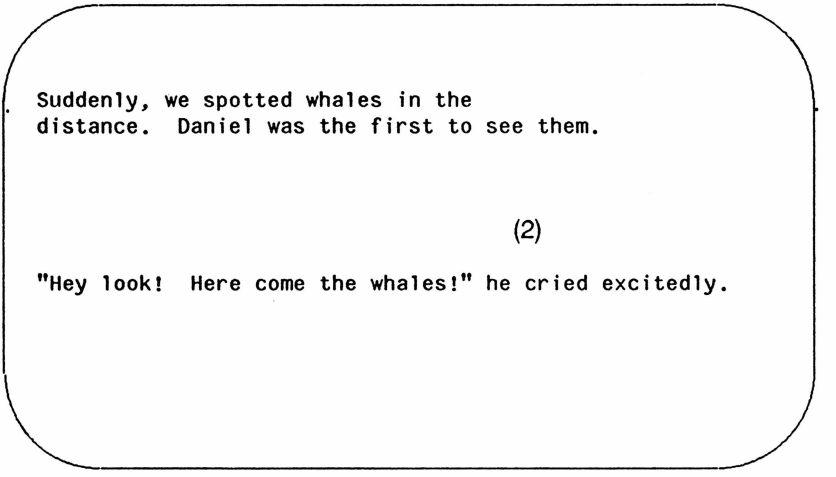
<N> Repeat the last search command in the opposite direction.

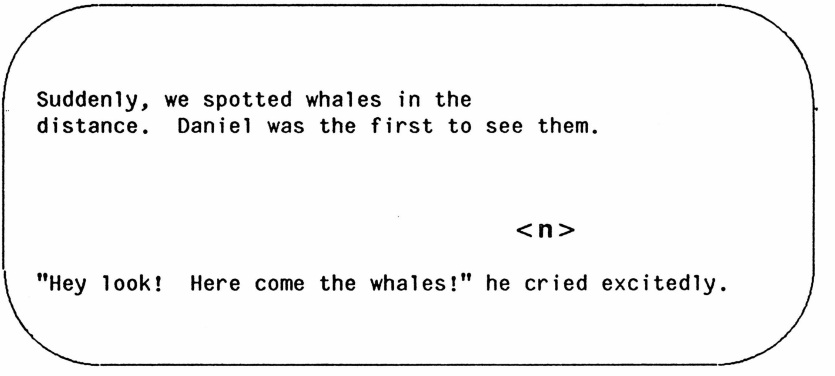
For example, say you want to search backward in the file for the three-letter pattern `the`. Initiate the search with **?***the* and continue it with **<n>**. The following screens offer a step-by-step illustration of how the **<n>** searches backward through the file and finds four occurrences of the character string `the`.

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.

"Hey look!  Here come the whales!" he cried excitedly.



?the
```


Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
.P
"Hey look! Here come the whales!" he cried excitedly.
(1)


Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
"Hey look! Here come the whales!" he cried excitedly.
<n>


Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
(2)
"Hey look! Here come the whales!" he cried excitedly.


Suddenly, we spotted whales in the
distance. Daniel was the first to see them.
<n>
"Hey look! Here come the whales!" he cried excitedly.

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.


                    (3)

"Hey look!  Here come the whales!" he cried excitedly.
```

```
Suddenly, we spotted whales in the
distance.  Daniel was the first to see them.


                    <n>
.P
"Hey look!  Here come the whales!" he cried excitedly.
```

```
Suddenly, we spotted whales in the


                                (4)
distance.  Daniel was the first to see them.
.P
"Hey look!  Here come the whales!" he cried excitedly.
```

The / and **?** search commands do not allow you to specify particular occurrences of a *pattern* with numbers. You cannot, for example, request the third occurrence (after your current position) of a *pattern*.

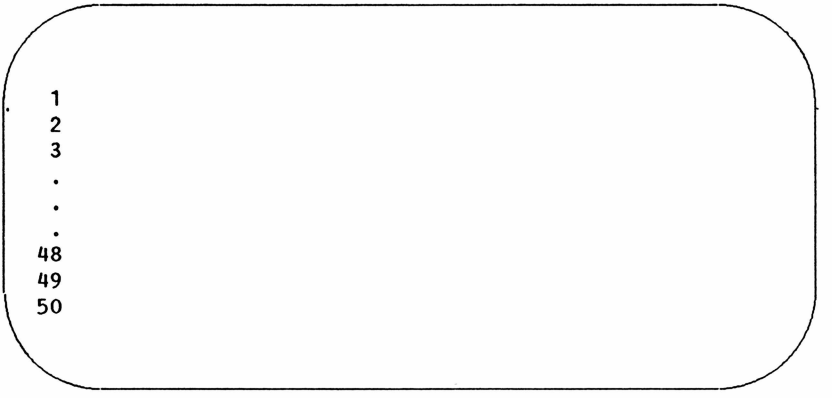
Figure 6-7 summarizes the **vi** commands for moving the cursor by scrolling the text, specifying a line number, and searching for a *pattern*.

Scrolling	
<^f>	Scroll the screen forward a full window, revealing the window of text below the current window.
<^d>	Scroll the screen down a half window, revealing lines below the current window.
<^b>	Scroll the screen back a full window, revealing the window of text above the current window.
<^u>	Scroll the screen up a half window, revealing the lines of text above the current window.
Positioning on a Numbered Line	
<1G>	Go to the first line of the file.
<G>	Go to the last line of the file.
<^g>	Give the line number and file status.
Searching for a Pattern	
/pattern	Search forward in the buffer for the next occurrence of the *pattern*. Position the cursor on the first character of the *pattern*.
?*pattern*	Search backward in the buffer for the first occurrence of the *pattern*. Position the cursor under the first character of the *pattern*.
<n>	Repeat the last search command.
<N>	Repeat the search command in the opposite direction.

Figure 6-7: Summary of Additional **vi** Motion Commands

Exercise 2

2-1. Create a file called **exer2**. Type a number on each line, numbering the lines from 1 to 50. Your file should look similar to the following.



2-2. Try using each of the scroll commands, noticing how many lines scroll through the window. Try the following:

<^f>
<^b>
<^u>
<^d>

2-3. Go to the end of the file. Append the following line of text.

```
123456789 123456789
```

What number does the command **<7h>** place the cursor on? What number does the command **<3l>** place the cursor on?

2-4. Try the command **<$>** and the command **<0>** (number zero).

2-5. Go to the first character on the line that is not a blank. Move to the first character in the next word. Move back to the first character of the word to the left. Move to the end of the word.

2-6. Go to the first line of the file. Try the commands that place the cursor in the middle of the window, on the last line of the window, and on the first line of the window.

2-7. Search for the number 8. Find the next occurrence of the number 8. Find 48.

Creating Text

There are three basic commands for creating text:

<a> append text

<i> insert text

<o> open a new line on which text can be entered

After you finish creating text with any one of these commands, you can return to the command mode of **vi** by pressing the ESCAPE key.

Appending Text

<a> append text after the cursor

<A> append text at the end of the current line

You have already experimented with the **<a>** command in the "Creating a File" section. Make a new file named **junk2**. Append some text using the **<a>** command. To return to command mode of **vi**, press the ESCAPE key. Then compare the **<a>** command to the **<A>** command.

Inserting Text

<i> insert text before the cursor

<I> insert text at the beginning of the current line before the first character that is not a blank

To return to the command mode of **vi**, press the ESCAPE key.

In the following examples you can compare the append and insert commands. The arrows show the position of the cursor, where new text will be added.

```
Append three spaces AFTER the H of Here

                                  <a>

Append three spaces AFTER the H of H   ere.

                                  <ESC>
```

```
Insert three spaces BEFORE the H of Here.

                                  <i>.

Insert three spaces BEFORE the H of   Here.

                                  <ESC>
```

Notice that in both cases, the user has left text input mode by pressing the ESCAPE key.

Opening a Line for Text

<o> Create text from the beginning of a new line below the current line. You can issue this command from any point in the current line.

<O> Create text from the beginning of a new line above the current line. This command can also be issued from any position in the current line.

The open command creates a directly above or below the current line, and puts you into text input mode. For example, in the following screens the **<O>** command opens a line above the current line, and the **<o>** command opens a line below the current line. In both cases, the cursor waits for you to enter text from the beginning of the new line.

```
   Create text ABOVE the current line.


                    <O>

[blank line]
   Create text ABOVE the current line.
```

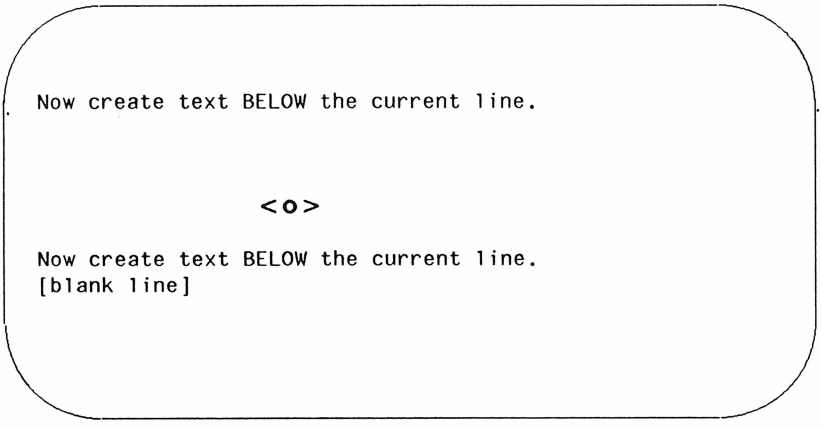


Figure 6-8 summarizes the commands for creating and adding text with the **vi** editor.

Command	Function
<a>	Create text after the cursor.
<A>	Create text at the end of the current line.
<i>	Create text in front of the cursor.
<I>	Create text before the first character on the current line that is not a blank.
<o>	Create text at the beginning of a new line below the current line.
<O>	Create text at the beginning of a new line above the current line.
<ESC>	Return **vi** to command mode from any of the above text input modes.

Figure 6-8: Summary of **vi** Commands for Creating Text

Exercise 3

3-1. Create a text file called **exer3**.

3-2. Insert the following four lines of text.

Append text
Insert text
a computer's
job is boring.

3-3. Add the following line of text above the last line:

financial statement and

3-4. Using a text insert command, add the following line of text above the third line:

Delete text

3-5. Add the following line of text below the current line:

byte of the budget

3-6. Using an append command, add the following line of text below the last line:

But, it is an exciting machine.

3-7. Move to the first line and add the word some before the word text.

Now practice using each of the six commands for creating text.

3-8. Leave **vi** and go on to the next section to find out how to delete any mistakes you made in creating text.

Deleting Text

You can delete text with various commands in command mode, and undo the entry of small amounts of text in text input mode. In addition, you can undo entirely the effects of your most recent command.

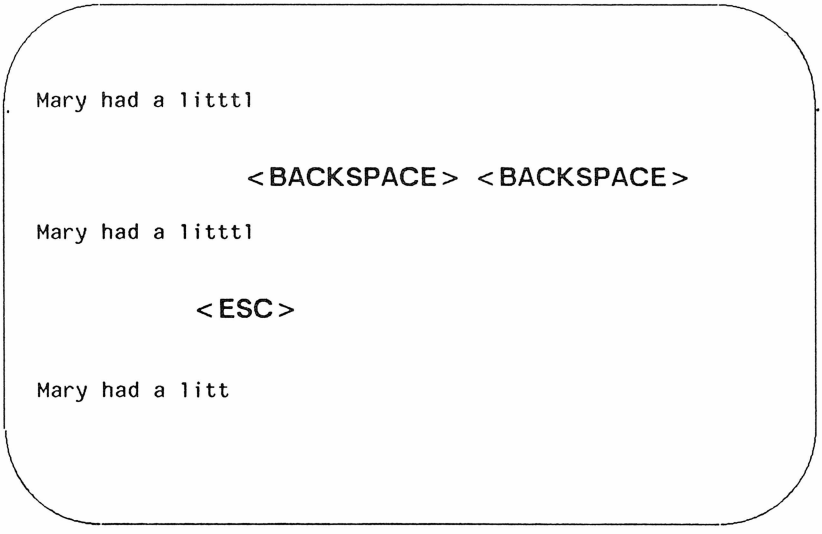
Undoing Entered Text in Text Input Mode

To delete a character at a time when you are in text input mode use the BACKSPACE key.

<BACKSPACE> Delete the current character (the character shown by the cursor).

The BACKSPACE key backs up the cursor in text input mode and deletes each character that the cursor backs across. However, the deleted characters are not erased from the screen until you type over them or press the ESCAPE key to return to command mode.

In the following example, the arrows represent the cursor.



Notice that the characters are not erased from the screen until you press the ESCAPE key.

There are two other keys that delete text in text input mode. Although you may not use them often, you should be aware that they are available. To remove the special meanings of these keys so that they can be typed as text, see the section on special commands.

<^w> undo the entry of the current word

<@> delete all text entered on current line since text input mode was entered

When you type <^w>, the cursor backs up over the word last typed and waits on the first character. It does not literally erase the word until you press the ESCAPE key or enter new characters over the old ones. The <@> sign behaves in a similar manner except that it removes all text you have typed on the current line since you last entered input mode.

Undo the Last Command

Before you experiment with the delete commands, you should try the **u** command. This command undoes the last command you issued.

<u> undo the last command

<U> restore the current line to its state before you changed it

If you delete lines by mistake, type **<u>**; your lines will reappear on the screen. If you type the wrong command, type **<u>** and it will be nullified. The **<U>** command will nullify all changes made to the current line as long as the cursor has not been moved from it.

If you type **<u>** twice in a row, the second command will undo the first; your undo will be undone! For example, say you delete a line by mistake and restore it by typing **<u>**. Typing **<u>** a second time will delete the line again. Knowing this command can save you a lot of trouble.

Delete Commands in Command Mode

You know that you can precede a command by a number. Many of the commands in **vi**, such as the delete and change commands, also allow you to enter a cursor movement command after another command. The cursor movement command can specify a text object such as a word, line, sentence, or paragraph. The general format of a **vi** command is:

[number][command]text_object

The brackets around some components of the command format show that those components are optional.

All delete commands issued in command mode immediately remove unwanted text from the screen and redraw the affected part of the screen.

The delete command follows the general format of a **vi** command.

[*number*]**d***text_object*

Deleting Words

You can delete a word or part of a word with the **<dw>** command. Move the cursor to the first character to be deleted and type **<dw>**. The character under the cursor and all subsequent characters in that word will be erased.

```
the deep dark depths of the lake.

   <2dw>
```

```
the depths of the lake.
```

The <**dw**> command deletes one word or punctuation mark and the space(s) that follow it. You can delete several words or marks at once by specifying a number before the command. For example, to delete three words and two commas, type <**5dw**>.

```
the deep, deep, dark depths of the lake

    <5dw>
```

```
the depths of the lake
```

Deleting Paragraphs

To delete paragraphs, use the following commands.

<**d{**> or <**d}**>

Observe what happens to your file. Remember, you can restore the deleted text with <**u**>.

Deleting Lines

To delete a line, type **<dd>**. To delete multiple lines, specify a number before the command. For example, typing

<10dd>

will erase ten lines. If you delete more than a few lines, **vi** will display this notice on the bottom of the screen:

```
10 lines deleted
```

If there are less than ten lines below the current line in the file, a bell will sound and no lines will be deleted.

Deleting Text After the Cursor

To delete all text on a line after the cursor, put the cursor on the first character to be deleted and type

<D> or **<d$>**.

Neither of these commands allows you to specify a number of lines; they can be used only on the current line.

Figure 6-9 summarizes the **vi** commands for deleting text.

Command	Function
For INSERT Mode:	
<BACKSPACE>	Delete the current character.
<^h>	Delete the current character.
<^W>	Delete the current word.
<@>	Delete the current line of new text or delete all new text on the current line.
For COMMAND Mode:	
<u>	Undo the last command.
<U>	Restore current line to its previous state.
<x>	Delete the current character.
<*n***d***x*>	Delete *n* number of text objects of type *x*.
<dw>	Delete the word at the cursor through the next space or to the next punctuation mark.
<dW>	Delete the word and punctuation at the cursor through the next space.
<dd>	Delete the current line.
<D>	Delete the portion of the line to the right of the cursor.
<d)>	Delete the current sentence.
<d}>	Delete the current paragraph.

Figure 6-9: Summary of Delete Commands

Exercise 4

4-1. Create a file called **exer4** and put the following four lines of text in it:

When in the course of human events
there are many repetitive, boring
chores, then one ought to get a
robot to perform those chores.

4-2. Move the cursor to line two and append to the end of that line:

tedious and unsavory.

Delete the word unsavory while you are in append mode.

Delete the word boring while you are in command mode.

What is another way you could have deleted the word boring?

4-3. Insert at the beginning of line four:

congenial and computerized.

Delete the line.

How can you delete the contents of the line without removing the line itself?

Delete all the lines with one command.

4-4. Leave the screen editor and remove the empty file from your directory.

Modifying Text

The delete commands and text input commands provide one way for you to modify text. Another way you can change text is by using a command that lets you delete and create text simultaneously. There are three basic change commands: **<r>**, **<s>**, and **<c>**.

Replacing Text

<r> Replace the current character (the character shown by the cursor). This command does not initiate text input mode, and so does not need to be followed by pressing the ESCAPE key.

<*n*r> Replace *n* characters with the same letter. This command automatically terminates after the *nth* character is replaced. It does not need to be followed by pressing the ESCAPE key.

<R> Replace only those characters typed over until the ESCAPE command is given. If the end of the line is reached, this command will append the input as new text.

The **<r>** command replaces the current character with the next character that is typed in. For example, suppose you want to change the word acts to ants in the following sentence:

```
The circus has many acts.
```

Place the cursor under the `c` of `acts` and type

<r>n

The sentence becomes

```
The circus has many ants.
```

To change `many` to `7777`, place the cursor under the `m` of `many` and type

<4r7>

The **<r>** command changes the four letters of `many` to four

occurrences of the number seven.

```
The circus has 7777 ants.
```

Substituting Text

The substitute command replaces characters, but then allows you to continue to insert text from that point until you press the ESCAPE key.

<s> Delete the character shown by the cursor and append text. End the text input mode by pressing the ESCAPE key.

<*n*s> Delete *n* characters and append text. End the text input mode by pressing the ESCAPE key.

<S> Replace all the characters in the line.

When you enter the **<s>** command, the last character in the string of characters to be replaced is overwritten by a `$` sign. The characters are not erased from the screen until you type over them, or leave text input mode by pressing the ESCAPE key.

Notice that you cannot use an argument with either **<r>** or **<s>**. Did you try?

Suppose you want to substitute the word million for the word hundred in the sentence `My salary is one hundred dollars.` Put the cursor under the `h` of `hundred` and type **<7s>**. Notice where the `$` sign appears.

```
My salary is one hundred dollars.

                <7s>
```

Then type **million**.

```
My salary is one hundre$ dollars.

                million
```

```
My salary is one million dollars.
```

Changing Text

The substitute command replaces characters. The change command replaces text objects, and then continues to append text from that point until you press the ESCAPE key. To end the change command, press the ESCAPE key.

The change command can take an argument. You can replace a character, word, or an entire line with new text.

<*n***c***x*>	Replace *n* number of text objects of type *x*, such as sentences (shown by <)>) and paragraphs (shown by <}>).
<**cw**>	Replace a word or the remaining characters in a word with new text. The **vi** editor prints a $ sign to show the last character to be changed.
<*n***cw**>	Replace *n* words.
<**cc**>	Replace all the characters in the line.
<*n***cc**>	Replace all characters in the current line and up to *n* lines of text.
<**C**>	Replace the remaining characters in the line, from the cursor to the end of the line.
<*n***C**>	Replace the remaining characters from the cursor in the current line and replace all the lines following the current line up to *n* lines.

The change commands, <**cw**> and <**C**>, use a $ sign to mark the last letter to be replaced. Notice how this works in the following example:

```
They are now due to arrive on Tuesday.

                                    <cw>
```

```
They are now due to arrive on Tuesda$.

                                    Wednesday<ESC>
```

```
They are now due to arrive on Wednesday.
```

Notice that the new word (Wednesday) has more letters than the word it replaced (Tuesday). Once you have executed the change

command you are in text input mode and can enter as much text as you want. The buffer will accept text until you press the ESCAPE key.

The **<C>** command, when used to change the remaining text on a line, works in the same way. When you enter the command it uses a $ sign to mark the end of the text that will be deleted, puts you in text input mode, and waits for you to type new text over the old. The following screens offer an example of the **C** command.

```
This is line 1.
Oh, I must have the wrong number.


<C>
This is line 3.
This is line 4.
```

```
This is line 1.
Oh, I must have the wrong number$


This is line 2.<ESC>
This is line 3.
This is line 4.
```

```
This is line 1.
This is line 2.
This is line 3.
This is line 4.
```

Now try combining arguments. For example, type

<c{>

Because you know the undo command, do not hesitate to experiment with different arguments or to precede the command with a number. You must press the ESCAPE key before using the **<u>** command, since **<c>** places you in text input mode.

Compare **<S>** and **<cc>**. The two commands should produce the same results.

Figure 6-10 summarizes the **vi** commands for changing text.

Command	Function
<r>	Replace the current character.
<R>	Replace only those characters typed over with new characters until the ESCAPE key is pressed.
<s>	Delete the character the cursor is on and append text. End the append mode by pressing the ESCAPE key.
<S>	Replace all the characters in the line.
<cc>	Replace all the characters in the line.
<*n***c***x*>	Replace *n* number of text objects of type *x*, such as sentences (shown by <**)**>) and paragraphs (shown by <**}**>).
<cw>	Replace a word or the remaining characters in a word with new text.
<C>	Replace the remaining characters in the line, from the cursor to the end of the line.

Figure 6-10: Summary of **vi** Commands for Changing Text

Cutting And Pasting Text Electronically

vi provides a set of commands that cut and paste text in a file. Another set of commands copies a portion of text and places it in another section of a file.

Moving Text

You can move text from one place to another in the **vi** buffer by deleting the lines and then placing them at the required point. The last text that was deleted is stored in a temporary buffer. If you move the cursor to that part of the file where you want the deleted lines to be placed and press the **p** key, the deleted lines will be added below the current line.

<p> Place the contents of the temporary buffer after the cursor.

A partial sentence that was deleted by the **<D>** command can be placed in the middle of another line. Position the cursor in the space between two words, then press **<p>**. The partial line is placed after the cursor.

Characters deleted by *<n***x**> also go into a temporary buffer. Any text object that was just deleted can be placed somewhere else in the text with **<p>**.

The **<p>** command should be used right after a delete command since the temporary buffer only stores the results of one command at a time. The **<p>** command is also used to copy text placed in the temporary buffer by the yank command. The yank command (**<y>**) is discussed in "Copying Text."

Fixing Transposed Letters

A quick way to fix transposed letters is to combine the **<x>** and the **<p>** commands as **<xp>**. **<x>** deletes the letter. **<p>** places it after next character.

Notice the error in the next line.

```
A line of tetx
```

This error can be changed quickly by placing the cursor under the `t` in `tx` and then pressing the **<x>** and **<p>** keys, in that order. The result is:

```
A line of text
```

Try this. Make a typing error in your file and use the **<xp>** command to correct it. Why does this command work?

Copying Text

You can yank (copy) one or more lines of text into a temporary buffer, and then put a copy of that text anywhere in the file. To put the text in a new position type **<p>**; the text will appear on the next line.

The yank command follows the general format of a **vi** command.

[*number*]**y**[*text_object*]

Yanking lines of text does not delete them from their original position in the file. If you want the same text to appear in more than one place, this provides a convenient way to avoid typing the same text several times. However, if you do not want the same text in multiple places, be sure to delete the original text after you have put the text into its new position.

Figure 6-11 summarizes the ways you can use the yank command.

Command	Function
<*n*y*x*>	Yank *n* number of text objects of type *x*, (such as sentences) and paragraphs }).
<yw>	Yank a copy of a word.
<yy>	Yank a copy of the current line.
<*n*yy>	Yank *n* lines.
<y)>	Yank all text up to the end of a sentence.
<y}>	Yank all text up to the end of the paragraph.

Figure 6-11: Summary of the Yank Command

Notice that this command allows you to specify the number of text objects to be yanked.

Try the following command lines and see what happens on your screen. (Remember, you can always undo your last command.) Type:

<5yw>

Move the cursor to another spot. Type:

<p>

Now try yanking a paragraph <y}> and placing it after the current paragraph. Then move to the end of the file <G> and place that same paragraph at the end of the file.

Copying or Moving Text Using Registers

Moving or copying several sections of text to a different part of the file is tedious work. **vi** provides a shortcut for this: named registers in which you can store text until you want to move it. To store text you can either yank or delete the text you wish to store.

Using registers is useful if a piece of text must appear in many places in the file. The extracted text stays in the specified register until you either end the editing session, or yank or delete another section of text to that register.

The general format of the command is:

[*number*]["*x*]*command*[*text_object*]

The *x* is the name of the register and can be any single letter. It must be preceded by a double quotation mark. For example, place the cursor at the beginning of a line. Type:

<3"ayy>

Type in more text and then go to the end of the file. Type:

<"ap>

Did the lines you saved in register **a** appear at the end of the file?

Figure 6-12 summarizes the cut and paste commands.

Command	Function
<**p**>	Place the contents of the temporary buffer containing the text obtained from the most recent delete or yank command into the text after the cursor.
<**yy**>	Yank a line of text and place it into a temporary buffer.
<*n***y***x*>	Yank a copy of *n* number of text objects of type *x* and place them in a temporary buffer.
<"**x**y*n*>	Place a copy of a text object of type *n* in the register named by the letter x.
<"**xp**>	Place the contents of the register x after the cursor.

Figure 6-12: Summary of **vi** Commands for Cutting and Pasting Text

Exercise 5

5-1. Enter **vi** with the file called **exer2**. that you created in Exercise 2.

Go to line eight and change its contents to **END OF FILE**

5-2. Yank the first eight lines of the file and place them in register **z**. Put the contents of register **z** after the last line of the file.

5-3. Go to line eight and change its contents to **eight is great**

5-4. Go to the last line of the file. Substitute **EXERCISE** for FILE Replace OF with TO

Special Commands

Here are some special commands that you will find useful.

<.>	repeat the last command
<J>	join two lines together
<^l>	clear the screen and redraw it
< >	change lower case to upper case and vice versa

Repeating the Last Command

The . period repeats the last command to create, delete, or change text in the file. It is often used with the search command.

For example, suppose you forget to capitalize the S in United States. However, you do not want to capitalize the s in chemical states. One way to correct this problem is by searching for the word states. The first time you find it in the expression United States, you can change the s to S. Then continue your search. When you find another occurrence, you can simply type a period; **vi** will remember your last command and repeat the substitution of s for S.

Experiment with this command. For example, if you try to add a period at the end of a sentence while in command mode, the last text change will suddenly appear on the screen. Watch the screen to see how the text is affected.

Joining Two Lines

The <J> command joins lines. To enter this command, place the cursor on the current line, and press the SHIFT and **j** keys simultaneously. The current line is joined with the following line.

For example, suppose you have the following two lines of text:

```
Dear Mr.
Smith:
```

To join these two lines into one, place the cursor under any character in the first line and type:

<J>

You will immediately see the following on your screen:

```
Dear Mr. Smith:
```

Notice that **vi** automatically places a space between the last word on the first line and the first word on the second line.

Clearing and Redrawing the Window

If another UNIX system user sends you a message using the write command while you are editing with **vi**, the message will appear in your current window, over part of the text you are editing. To restore your text after you have read the message, you must be in command mode. (If you are in text input mode, press the ESCAPE key to return to command mode.) Then type **<^l>** (control-l). **vi** will erase the message and redraw the window exactly as it appeared before the message arrived.

Changing Lower Case to Upper Case and Vice Versa

A quick way to change any lower case letter to upper case, or vice versa, is by putting the cursor on the letter to be changed and typing a < > (tilde). For example, to change the letter a to A, press . You can change several letters by typing several times, but you cannot precede the command with a number to change several letters with one command.

Figure 6-13 summarizes the special commands.

Command	Function
<.>	Repeat the last command.
<J>	Join the line below the current line with the current line.
<^l>	Clear and redraw the current window.
< >	Change lower case to upper case, or vice versa.

Figure 6-13: Summary of Special Commands

Using Line Editing Commands in vi

The **vi** editor has access to many of the commands provided by a line editor called **ex**. (For a complete list of **ex** commands see the **ex**(1) page in the *User's Reference Manual*.) This section discusses some of those most commonly used.

The **ex** commands are very similar to the **ed** commands discussed in Chapter 5. If you are familiar with **ed**, you may want to experiment on a test file to see how many **ed** commands also work in **vi**.

Line editor commands begin with a **:** (colon). After the colon is typed, the cursor will drop to the bottom of the screen and display the colon. The remainder of the command will also appear at the bottom of the screen as you type it.

Temporarily Returning to the Shell: the :sh and :! Commands

When you enter **vi**, the contents of the buffer fill your screen, making it impossible to issue any shell commands. However, you may want to do so. For example, you may want to get information from another file to incorporate into your current text. You could get that information by running one of the shell commands that display the text of a file on your screen, such as the **cat** or **pg** command. However, quitting and reentering the editor is time consuming and tedious. **vi** offers two methods of escaping the editor temporarily so that you can issue shell commands (and even edit other files) without having to write your buffer and quit: the **:!** command and the **:sh** command.

The **:!** command allows you to escape the editor and run a shell command on a single command line. From the command mode of **vi**, type **:!**. These characters will be printed at the bottom of your screen. Type a shell command immediately after the **!**. The shell will run your command, give you output, and print the message `[Hit return to continue]`. When you press the RETURN key **vi** will refresh the screen and the cursor will reappear exactly where you left it.

The **ex** command **:sh** allows you to do the same thing, but behaves differently on the screen. From the command mode of **vi** type **:sh** and press the RETURN key. A shell command prompt will appear on the next line. Type your command(s) after the prompt as you would normally do while working in the shell. When you are ready to return to **vi**, type <^**d**> or **exit**; your screen will be refreshed with your buffer contents and the cursor will appear where you left it.

Even changing directories while you are temporarily in the shell will not prevent you from returning to the **vi** buffer where you were editing your file when you type **exit** or <^**d**>.

Writing Text to a New File: the :w Command

The **:w** (for write) command allows you to create a file by copying lines of text from the file your are currently editing into a file that you specify. To create your new file you must specify a line or range of lines (with their line numbers), along with the name of the new file, on the command line. You can write as many lines as you like. The general format is:

:*line_number[,line_number]***w** *filename*

For example, to write the third line of the buffer to a line named **three**, type:

:3w three<**CR**>

vi reports the successful creation of your new file with the following information:

```
"three" [New file] 1 line, 20 characters
```

To write your current line to a file, you can use a **.** (period) as the line address:

:.w junk<**CR**>

A new file called **junk** will be created. It will contain only the current line in the **vi** buffer.

You can also write a whole section of the buffer to a new file by specifying a range of lines. For example, to write lines 23 through 37 to a file, type the following:

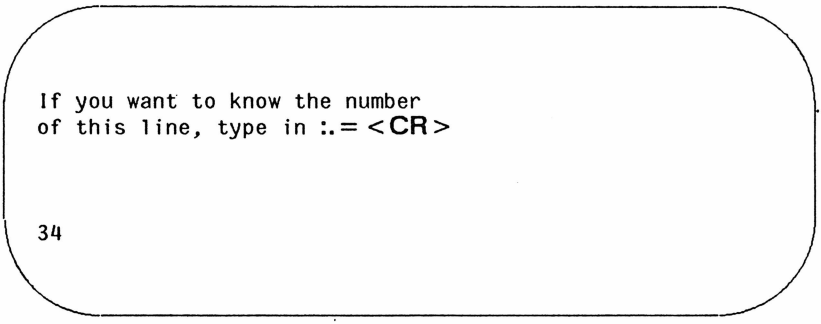
:23,37w *newfile* **<CR>**

Finding the Line Number

To determine the line number of a line, move the cursor to it and type **:** (colon). The colon will appear at the bottom of the screen. Type **.=** after it and press the RETURN key.

```
If you want to know the number
of this line, type :.= <CR>


:.=
```

As soon as you press the RETURN key, your command line will disappear from the bottom line and be replaced by the number of your current line in the buffer.



You can move the cursor to any line in the buffer by typing **:** and the line number. The command line

:*n* **<CR>**

means to go to the *n*th line of the buffer.

Deleting the Rest of the Buffer

One of the easiest ways to delete all the lines between the current line and the end of the buffer is by using the line editor command **d** with the special symbols for the current and last lines.

:.,$d <CR>

The **.** represents the current line; the **$** sign, the last line.

Adding a File to the Buffer

To add text from a file below a specific line in the editing buffer, use the **:r** (read) command. For example, to put the contents of a file called **data** into your current file, place the cursor on the line above the place where you want it to appear. Type:

:r data <CR>

You may also specify the line number instead of moving the cursor. For example, to insert the file **data** below line 56 of the buffer, type

:56r data<CR>

Do not be afraid to experiment; you can use the **<u>** command to undo **ex** commands, too.

Making Global Changes

One of the most powerful commands in **ex** is the global command. The global command is given here to help those users who are familiar with the line editor. Even if you are not familiar with a line editor, you may want to try the command on a test file.

For example, say you have several pages of text about the DNA molecule in which you refer to its structure as a helix. Now you want to change every occurrence of the word helix to double helix. The **ex** editor's global command allows you to do this with one command line. First, you need to understand a series of commands.

:g/*pattern*/*command*<**CR**>

For each line containing *pattern*, execute the **ex** command named *command*. For example, type: **:g/helix<CR>**. The line editor will print all lines that contain the pattern helix.

:s/*pattern*/*new_words*/<**CR**>

This is the substitute command. The line editor searches for the first instance of the characters *pattern* on the current line and changes them to *new_words*.

:s/*pattern*/*new_words*/**g**<**CR**>

If you add the letter g after the last delimiter of this command line, **ex** will change every occurrence of *pattern* on the current line. If you do not, **ex** will change only the first occurrence.

:g/*helix*/**s**//*double helix*/**g** < **CR** >

This command line searches for the word helix. Each time helix is found, the substitute command substitutes two words, double helix, for every instance of helix on that line. The delimiters after the **s** do not need to have helix typed in again. The command remembers the word from the delimiters after the global command **g**. This is a powerful command. For a more detailed explanation of global and substitution commands, see Chapter 5.

Figure 6-14 summarizes the line editor commands available in **vi**.

Command	Function
:	Shows that the commands that follow are line editor commands.
:sh <CR>	Temporarily returns you to the shell to perform shell commands.
<^d>	Escapes the temporary shell and returns you to the current window of **vi** to continue editing.
:n **<CR>**	Goes to the *n*th line of the buffer.
:*x,y***w** *data* **<CR>**	Writes lines from the number *x* through the number *y* into a new file (*data*).
:$ <CR>	Goes to the last line of the buffer.
:.,$d <CR>	Deletes all the lines in the buffer from the current line to the last line.
:r *shell.file* **<CR>**	Inserts the contents of *shell.file* after the current line of the buffer.
:s/*text*/*new_words*/ **<CR>**	Replaces the first instance of the characters *text* on the current line with *new_words*.
:s/*text*/*new_words*/**g <CR>**	Replaces every occurrence of *text* on the current line with *new_words*.
:g/*text*/**s**//*new_words*/**g <CR>**	Replaces every occurrence of *text* in the file with *new_words*.

Figure 6-14: Summary of Line Editor Commands

Quitting vi

There are five basic command sequences to quit the **vi** editor. Commands that are preceded by a colon (:) are line editor commands.

<ZZ> or **:wq<CR>**	Write the contents of the **vi** buffer to the UNIX file currently being edited and quit **vi**.
:w *filename*<*CR*> **:q<CR>**	Write the temporary buffer to a new file named *filename* and quit **vi**.
:w! *filename*<*CR*> **:q<CR>**	Overwrite an existing file called *filename* with the contents of the buffer and quit **vi**.
:q!<CR>	Quit **vi** without writing the buffer to a file, and discard all changes made to the buffer.
:q<CR>	Quit **vi** without writing the buffer to a UNIX file. This works only if you have made no changes to the buffer; otherwise **vi** will warn you that you must either save the buffer or use the **:q!<CR>** command to terminate.

The **<ZZ>** command and **:wq** command sequence both write the contents of the buffer to a file, quit **vi**, and return you to the shell. You have tried the **<ZZ>** command. Now try to exit **vi** with **:wq**. **vi** remembers the name of the file currently being edited, so you do not have to specify it when you want to write the buffer's contents back into the file. Type

:wq<CR>

The system responds in the same way it does for the **<ZZ>** command. It tells you the name of the file, and reports the number of lines and characters in the file.

What must you do to give the file a different name? For example, suppose you want to write to a new file called **junk**. Type:

:w junk<CR>

After you write to the new file, leave **vi**. Type:

:q<CR>

If you try to write to an existing file, you will receive a warning. For example, if you try to write to a file called **johnson**, the system will respond with:

```
"johnson" File exists - use "w! johnson" to overwrite
```

If you want to replace the contents of the existing file with the contents of the buffer, use the **:w!** command to overwrite **johnson**.

:w! johnson<CR>

Your new file will overwrite the existing one.

If you edit a file called **memo**, make some changes to it, and then decide you don't want to keep the changes, or if you accidentally press a key that gives **vi** a command you cannot undo, leave **vi** without writing to the file. Type:

:q!<CR>

Figure 6-15 summarizes the quit commands.

Command	Function
<ZZ>	Write the file and quit **vi**.
:wq<CR>	Write the file and quit **vi**.
:w *filename* **<CR>** **:q<CR>**	Write the editing buffer to a new file (*filename*) and quit **vi**.
:w! *filename* **<CR>** **:q<CR>**	Overwrite an existing file (*filename*) with the contents of the editing buffer and quit **vi**.
:q!<CR>	Quit **vi** without writing buffer to a file.
:q<CR>	Quit **vi** without writing the buffer to a file.

Figure 6-15: Summary of the Quit Commands

Special Options For vi

The **vi** command has some special options. It allows you to:

- recover a file lost by an interrupt to the UNIX system
- place several files in the editing buffer and edit each in sequence, and
- view the file at your own pace by using the **vi** cursor positioning commands

Recovering a File Lost by an Interrupt

If there is an interrupt or disconnect, the system will exit the **vi** command without writing the text in the buffer back to its file. However, the UNIX system will store a copy of the buffer for you. When you log back in to the UNIX system you will be able to restore the file with the **–r** option for the **vi** command. Type

vi –r *filename* **<CR>**

The changes you made to *filename* before the interrupt occurred are now in the **vi** buffer. You can continue editing the file, or you can write the file and quit **vi**. The **vi** editor will remember the file name and write to that file.

Editing Multiple Files

If you want to edit more than one file in the same editing session, issue the **vi** command, specifying each file name. Type

vi *file1 file2* **<CR>**

vi responds by telling you how many files you are going to edit. For example:

```
2 files to edit
```

After you have edited the first file, write your changes (in the buffer) to the file (*file1*). Type

:w <CR>

The system response to the **:w <CR>** command will be a message at the bottom of the screen giving the name of the file, and the number of lines and characters in that file. Then you can bring the next file into the editing buffer by using the **:n** command. Type

:n <CR>

The system responds by printing a notice at the bottom of the screen, telling you the name of the next file to be edited and the number of characters and lines in that file.

Select two of the files in your current directory. Then enter **vi** and place the two files in the editing buffer at the same time. Notice the system responses to your commands at the bottom of the screen.

Viewing a File

It is often convenient to be able to inspect a file by using **vi**'s powerful search and scroll capabilities. However, you might want to protect yourself against accidentally changing a file during an editing session. The read-only option prevents you from writing in a file. To avoid accidental changes, you can set this option by invoking the editor as **view** rather than **vi**.

Figure 6-16 summarizes the special options for **vi**.

Option	Function
vi *file1 file2 file3* <**CR**>	Enter three files (*file1*, *file2*, and *file3*) into the **vi** buffer to be edited.
:w <**CR**> **:n** <**CR**>	Write the current file and call the next file into the buffer.
vi **−r** *file1* <CR>	Restore the changes made to *file1*.

Figure 6-16: Summary of Special Options for **vi**

Exercise 6

6-1. Try to restore a file lost by an interrupt.

Enter **vi**, create some text in a file called **exer6**. Turn off your terminal without writing to a file or leaving **vi**. Turn your terminal back on, and log in again. Then try to get back into **vi** and edit **exer6**.

6-2. Place **exer1** and **exer2** in the **vi** buffer to be edited. Write **exer1** and call in the next file in the buffer, **exer2**.

Write **exer2** to a file called **junk**.

Quit **vi**.

6-3. Try out the command:

vi exer* <CR>

What happens? Try to quit all the files as quickly as possible.

6-4. Look at **exer4** in read-only mode.

Scroll forward.

Scroll down.

Scroll backward.

Scroll up.

Quit and return to the shell.

Answers To Exercises

There is often more than one way to perform a task in **vi**. Any method that works is correct. The following are suggested ways of doing the exercises.

Exercise 1

1-1. Ask your system administrator for your terminal's system name. Type:

TERM = *terminal_name* < **CR** >

1-2. Enter the **vi** command for a file called **exer1**:

vi exer1 < **CR** >

Then use the append command (< **a** >) to enter the following text in your file:

This is an exercise! < **CR** >
Up, down < **CR** >
left, right, < **CR** >
build your terminal's < **CR** >
muscles bit by bit < **ESC** >

1-3. Use the < **k** > and < **h** > commands.

1-4. Use the < **x** > command.

1-5. Use the <**j**> and <**l**> commands.

1-6. Enter **vi** and use the append command (<**a**>) to enter the following text:

and byte by byte<ESC>

Then use <**j**> and <**l**> to move to the last line and character of the file. Use the <**a**> command again to add text. You can create a new line by pressing the RETURN key. To leave text input mode, press the ESCAPE key.

1-7. Type:

<ZZ>

1-8. Type:

vi exer1<CR>

Notice the system response:

```
"exer1" 7 lines, 102 characters
```

Exercise 2

2-1. Type:

vi exer2<CR>
<a>1<CR>
2<CR>
3<CR>
.
.
.
48<CR>
49<CR>
50<ESC>

2-2. Type:

<^f>
<^b>
<^u>
<^d>

Notice the line numbers as the screen changes.

2-3. Type:

<G>
<o>
123456789 123456789<ESC>
<7h>
<3l>

Typing **<7h>** puts the cursor
on the 2 in the second set of numbers.
Typing **<3l>** puts the cursor
on the 5 in the
second set of numbers.

2-4. **$** = end of line
0 = first character in the line

2-5. Type:

<^>
<w>
<b>
<e>

2-6. Type:

<1G>
<M>
<L>
<H>

2-7. Type:

/8
<n>
/48

Exercise 3

3-1. Type:

vi exer3 <CR>

3-2. Type:

<a> Append text <CR>
Insert text <CR>
a computer's <CR>
job is boring. <ESC>

3-3. Type:

<O>
financial statement and <ESC>

3-4. Type:

<3G>
<i> Delete text <CR> <ESC>

The text in your file now reads:

```
Append text
Insert text
Delete text
a computer's
financial statement and
job is boring.
```

3-5. The current line is `a computer's`. To create a line of text below that line use the **<o>** command.

3-6. The current line is `byte of the budget.`
<G> puts you on the bottom line.
<A> lets you begin appending at the end of the line.
<CR> creates the new line.
Add the sentence: **But, it is an exciting machine.**
<ESC> leaves append mode.

3-7. Type:

<1G>
/text
<i> some <space bar> <ESC>

3-8. <ZZ> will write the buffer to **exer3** and return you to the shell.

Exercise 4

4-1. Type:

vi exer4<CR>
<a> When in the course of human events
<CR> there are many repetitive, boring<CR>
chores, then one ought to get a<CR>
robot to perform those chores.<ESC>

4-2. Type:

<2G>
<A> tedious and unsavory<8BACKSPACE>
<CR><ESC>

Press **<h>** until you get to the `b` of `boring`. Then type: **<dw>**. (You can also use **<6x>**.)

4-3. You are at the second line. Type:

<2j>
<I> congenial and computerized<ESC>
<dd>

To delete the line and leave it blank, type in:

<0> (zero moves the cursor to the beginning of the line)
<D>

<H>
<3dd>

4-4. Write and quit **vi**.

<ZZ>

Remove the file.

rm exer4<CR>

Exercise 5

5-1. Type:

vi exer2 < CR >
< 8G >
< cc > END OF FILE < ESC >

5-2. Type:

< 1G >
< 8"zyy >
< G >
< "zp >

5-3. Type:

< 8G >
< cc > 8 is great < ESC >

5-4. Type:

< G >
< 2w >
< cw >
EXERCISE < ESC >
< 2b >
< cw >
TO < ESC >

Exercise 6

6-1. Type:

vi exer6 < CR >
< a > (append several lines of text)
< ESC >

Turn off the terminal.

Turn on the terminal.
Log in on your UNIX system. Type:

vi −r exer6 < CR >

:wq <CR>

6-2. Type:

vi exer1 exer2 <CR>
:w <CR>
:n <CR>

:w junk <CR>
<ZZ>

6-3. Type:

vi exer* <CR>

(Response:)
`8 files to edit` (**vi** calls all files with names that begin with **exer**.)

<ZZ>
<ZZ>

6-4. Type:

view exer4 <CR>
<^f>
<^d>
<^b>
<^u>
:q <CR>

Substituting on One Line

To substitute text on a line that is not the current line, include an address in the command line, as follows:

[*address1*]**s**/*old_text*/*new_text*/ < **CR** >

For example, in the following screen the command line includes an address for the line to be changed (line 1) because the current line is line 3:

```
1,3p<CR>
This is a pest toad
testing testing
come in toad
.<CR>
come in toad
1s/pest/test<CR>
This is a test toad
```

As you can see, **ed** printed the new line automatically after the change was made, because the last delimiter was omitted.

Substituting on a Range of Lines

You can make a substitution on a range of lines by specifying the first address (*address1*) through the last address (*address2*).

[*address1*,*address2*]**s**/*old_text*/*new_text*/ < **CR** >

If **ed** does not find the pattern to be replaced on a line, no changes are made to that line.

In the following example, all the lines in the file are addressed for the substitute command. However, only the lines that contain the string **es** (the *old_text* argument) are changed.

```
1,$p<CR>
This is a test toad
testing testing
come in toad
testing 1, 2, 3
1,$s/es/ES/n<CR>
4        tESting 1, 2, 3
```

When you specify a range of lines and include **p** or **n** at the end of the substitute line, only the last line changed is printed.

To display all the lines in which text was changed, use the **n** or **p** command with the address **1,$**.

```
1,$n<CR>
1        This is a tESt toad
2        tESting testing
3        come in toad
4        tESting 1, 2, 3
```

Notice that only the first occurrence of **es** (on line 2) has been changed. To change every occurrence of a pattern, use the **g** command, described in the next section.